Práctica 6b:
Lectura de datos analógicos usando el RP240 y la Raspberry Pi

Fundamentos de Sistemas Embebidos

Autor: José Mauricio Matamoros de Maria y Campos

1. Objetivo

El alumno aprenderá a comunicar circuitos integrados mediante el bus I^2C, así como a leer e interpretar señales analógicas con un microcontrolador.

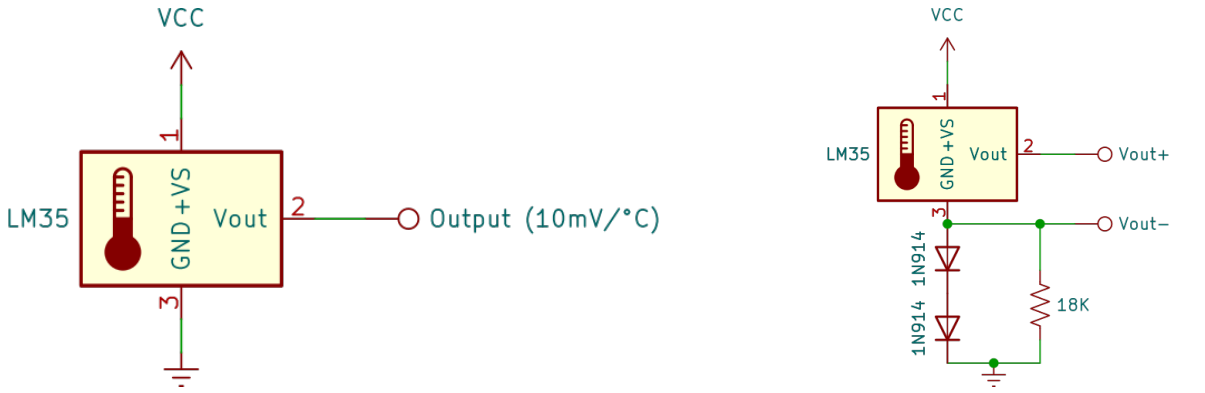
2. Introducción

La presente práctica resume los pasos a seguir para leer una señal analógica con un microcontrolador. En particular, se interesa en la lectura de la temperatura registrada por un sensor LM35 mediante un microcontrolador RP2040 (Raspberry Pico). Los datos registrados serán posteriormente enviados vía I^2C a una Raspberry Pi para llevar una bitácora de temperatura que podrá ser desplegada en un navegador web.

2.1. El sensor LM35

El circuito integrado LM35 es un sensor de temperatura cuya salida de voltaje o respuesta es linealmente proporcional a la temperatura registrada en escala centígrada. Una de las principales ventajas del LM35 sobre otros sensores lineales calibrados en Kelvin, es que no se requiere restar constantes grandes para obtener la temperatura en grados centígrados. El rango de este sensor va de -55°C a 150°C con una precisión que varía entre 0.5°C y 1.0°C dependiendo la temperatura medida [1].

Las configuraciones más comunes para este integrado se muestran en la Figura 1. La configuración (Figura 1a) básica, la más simple posible pues sólo requiere conectar al integrado LM35 entre Vcc y Gnd, permite medir temperaturas entre 2°C a 150°C. Por otro lado, la configuración (Figura 1a) clásica permite medir en todo el rango completo del sensor, es decir entre -55°C y 150°C, pero requiere de un par de diodos 1N914 y una resistencia de



(a) Configuración básica: 2°C a 150°C

(b) Configuración clásica: -55°C a 150°C

Figura 1: Configuraciones típicas del LM35

18KΩ para proporcionar los voltajes de referencia. En ambos casos, el LM35 ofrece una diferencial de $10mV/^oC$, por lo que los voltajes medidos rara vez excederán de 2V respecto a tierra.

Cuando opera en rango completo y las temperaturas registradas son inferiores a cero, se permite un flujo de corriente inverso entre los pines GND y V_{out} del LM35, es decir, una salida de voltaje negativo respecto a la referencia. Debido a que el LM35 no puede generar voltajes inferiores respecto a la referencia del circuito (tierra) se utilizan dos diodos 1N914 en serie colocados en el pin de referencia o tierra del LM35 (véase Figura 1b) para elevar el voltaje del subcircuito del LM35 aproximadamente 1.2V por encima del voltaje de referencia o tierra general. Así, cuando el LM35 entre en contacto con temperaturas negativas, el voltaje de diodo o V_{DD} referenciable mediante la resistencia de 18K hará posible que el voltaje de V_{out+} sea inferior al de V_{out-} y pueda calcularse la diferencia, tal como se muestra en la Tabla 1.

2.2. Convertidor Analógico—Digital

Para leer la señal del LM35 se requiere de un Convertidor Analógico Digital o ADC (por sus siglas en inglés: *Digital-Analog Converter*). Un ADC se elige con base en dos factores clave: su precisión y su tiempo de muestreo. Debido a que la aplicación del ADC será convertir mediciones de temperatura y los cambios de temperatura son muy lentos,[1] puede obviarse el tiempo de muestreo. En cuanto a la precisión, los convertidores A/D más comunes son de 8 y 10 bits, de los cuales ha de elegirse uno.

Tabla 1: Salida de un LM35 en rango completo

Temp [°C]	V_{out+} [V]
-55	0.65
0	1.20
50	1.70
100	2.20
150	2.70

La precisión del ADC se calcula tomando en cuenta el rango de operación y la precisión del componente analógico a discretizar. El LM35 tiene un rango de 205°C, una diferencial de voltaje $\Delta V = 10mV/^oC$ y una precisión máxima de 0.5°C, por lo que el sensor entregará un máximo de 2.5V respecto al voltaje de referencia del mismo, con incrementos de 5mV. Debido a que 256 valores para un rango de 205°C en incrementos de 0.5°C (es decir 410 valores) es claramente insuficiente para este sensor, por lo que será conveniente utilizar un convertidor A/D de 10 bits.

Un ADC típico de 12 bits convertirá las señales analógicas entre voltajes de referencia V_{Ref-} y V_{Ref+} como un entero con valores entre 0 y 4096, interpretando los valores V_{Ref-} como 0 lógico y V_{Ref+} como 4096 de manera aproximadamente lineal. El decir, la lectura obtenida es directamente proporcional al voltaje dentro del rango, estimable mediante la fórmula:

$$V_{out} = value \times \frac{V_{Ref+} - V_{Ref-}}{4096} \tag{1}$$

En una configuración simple, V_{Ref-} y V_{Ref+} se conectan internamente dentro del RP2040 a tierra y V_{CC} respectivamente. Esto simplifica la fórmula como:

$$V_{out} = value \times \frac{5V}{4096} = value \times 1.22mV \tag{2}$$

En lo concerniente al RP040, éste incorpora un convertidor analógico-digital de 12 bits con soporte para voltaje de referencia V_{Ref+}, denominado ADC_VREF según las especificaciones del mismo [2]. Considerando que el LM35 en rango completo entrega hasta 2.05V ($10mV \times (150 - -55) = 2.05V$) la mayor parte de los 4096 valores jamás serán ocupados. Por este motivo, conviene sacar partido del pin de voltaje de referencia ADC_VREF del RP2040 mediante un divisor de voltaje (véase Figura 2). En consecuencia, el pin ADC_VREF requerirá de un divisor de voltaje con salida de 2.73V tal como se muestra en la Figura 2 para dar mayor precisión al convertidor A/D.

Con esta nueva configuración, se puede calcular de nueva cuenta la precisión del sensor digital una vez decodificado el valor analógico leído del LM35 dividiendo los 2.73V de referencia entre los 4096 valores posibles que entrega el ADC como sigue:

$$\Delta V = \frac{2.73V}{4096} = 666.5 \times 10^{-6}V = 667\mu V \tag{3}$$

Debido a que la resolución máxima del sensor LM35 determinada por su factor de incertidumbre es de 0.5°C equivalentes a 0.005V, ambas configuraciones (con y sin el divisor de voltaje) serán adecuadas para operar al sensor.

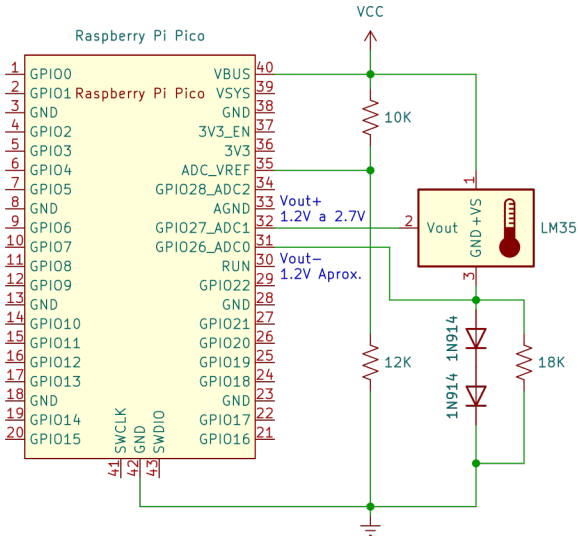


Figura 2: Circuito medidor de temperatura LM35 con el RP2040

2.3. Bus I²C

I²C es un protocolo serial inventado por Phillips y diseñado para conectar dispositivos de baja velocidad mediante interfaces de dos hilos (Figura 3). El protocolo permite un número virtualmente ilimitado de dispositivos interconectados donde más de uno puede ser un dispositivo maestro. El bus I2C es popular debido a su facilidad de uso y fácil configuración. Sólo es necesario definir la velocidad máxima del bus, que está conformado por dos cables con resistencias pull-up [3].

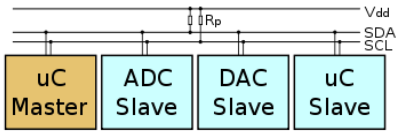


Figura 3: Bus I²C

I²C utiliza solamente dos cables: SCL (reloj) y SDA (datos). La transferencia de datos es serial y transmite paquetes de 8 bits con velocidades de hasta 5MHz. Además, es requisito que cada dispositivo esclavo tenga una dirección de 7 bits que (el bit más significativo se utiliza para indicar si el paquete es una lectura o una escritura) debe ser única en el bus. Los dispositivos maestros no necesitan dirección ya que estos generan la señal de reloj y coordinan a los dispositivos esclavos [3].

3. Material

Se asume que el alumno cuenta con un una Raspberry Pi con sistema operativo Raspbian e interprete de Python instalado. Se aconseja encarecidamente el uso de *git* como programa de control de versiones.

- 1 microcontrolador RP2040 (Raspberry Pico) con firmware MicroPython precargado.
- 1 sensor de temperatura LM35 en encapsulado TO-220 o TO-92
- 2 Diodos 1N914
- 2 resistencia de 10kΩ
- 1 resistencia de 12kΩ[2]
- 1 resistencia de 18kΩ
- 1 Condensador de 0.1μF
- 1 protoboard o circuito impreso equivalente
- 1 fuente de alimentación regulada a 5V y al menos 2 amperios de salida
- 1 cable micro-USB/USB-C para programar el RP2040
- Cables y conectores varios

[2]La resistencia de 12kΩ puede reemplazarse con resistencias de 13kΩ a 20kΩ dependiendo del voltaje de los diodos.

4. Instrucciones

1. Alambre el circuito mostrado en la Figura 2.
2. Realice los programas de las Subsecciones 4.3 y 4.4
3. Analice los programas de las subsecciones 4.3 y 4.4, realice los experimentos propuestos en la sección 5.

4.1. Paso 1: Alambrado

Importante

Aunque el RP2040 cuenta con un regulador de 5V integrado, opera a 3.3V. Asegúrese de conectar Vcc únicamente a V_{BUS} (Pin 38) para no quemarlo.

El proceso de alambrado de esta práctica considera dos circuitos. El primer circuito, mostrado en la Figura 2, permite obtener valores discretos del sensor de temperatura LM35. El segundo circuito (Figura 4) consiste en la interfaz de conexión vía I^2C entre el microcontrolador que lee el LM35 y la Raspberry Pi que genera los reportes y grafica los resultados.

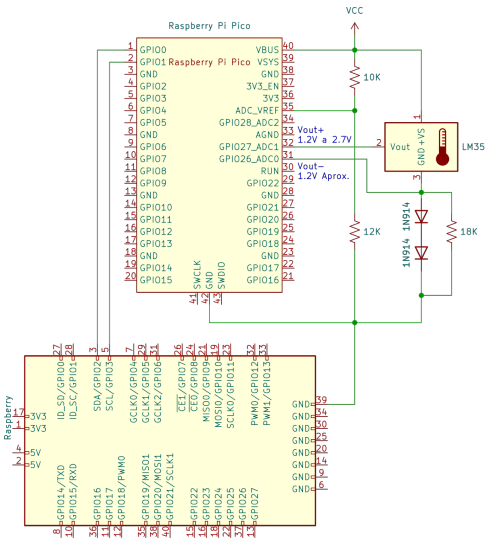


Figura 4: Circuito completo

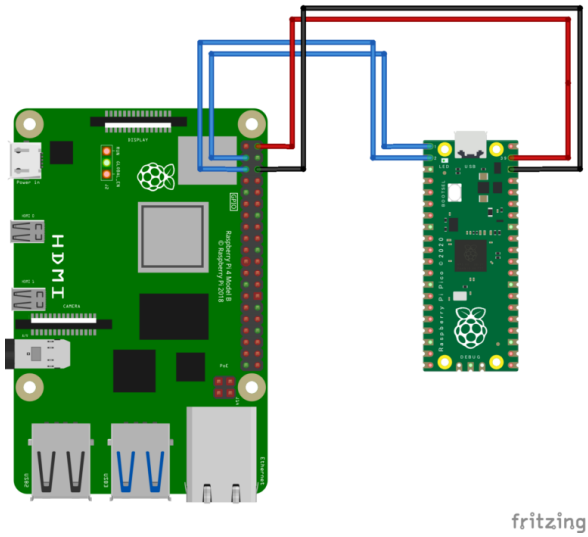


Figura 5: Conexión mediante I^2C de una Raspberry Pi con una Raspberry Pico[3]

Alambre primero el subcircuito formado por los dos diodos, el integrado LM35 y la resistencia de 18kΩ. Paso seguido, alimente el subcircuito y mida la diferencia de potencial existente entre V_{OUT-} y Gnd. Utilice el valor medido en la fórmula $V_{ADC\_VREF} = 1.5V + V_{OUT-}$ para calcular los valores de las resistencias que se conectarán al pin ADC_VREF del RP2040.

Importante

Asegúrese de que $V_{ADC\_VREF} \leq V_{OUT-}\Big|_{Temp=150^oC}$ para evitar quemar el RP2040.

Continúe el alambrado del circuito. Es conveniente colocar un capacitor de 0.1μF entre Vcc y Gnd para rectificar el voltaje de entrada eliminar cualquier oscilación parásita que pudiere afectar el funcionamiento del LM35. La presencia de este componente es opcional pero altamente recomendada.

[3]Imagen obtenida de https://python-academia.com/en/raspberry-pi-pico-slave/

Tabla 2: Conexiones I^{2}C entre Raspberry Pi y un RP2040

Pin Raspberry		Conexión			Pin RP2040	
3	(GPIO2)	Raspberry Pi SDA	→	RP2040 SDA	GPIO0/SDA	1
5	(GPIO3)	Raspberry Pi SCL	→	RP2040 SCL	GPIO1/SCL	2
6	(GND)	Raspberry Pi GND	→	RP2040 GND	GND	3

Tras alambrar el primer circuito realice el experimento prueba indicado en la Subsección 4.2.

A continuación conecte el bus I^{2}C entre la Raspberry Pi y el RP2040 como ilustran la Tabla 2 y la Figura 5.

Esto es posible debido a que el RP2040 no cuenta con resistencias de acoplamiento a positivo o *pull-up* integradas, mientras que los pines I^{2}C de la Raspberry Pi están conectados internamente a la línea de 3.3V mediante resistencias de 1.8kΩ. Por este motivo, tendrán que quitarse las resistencias de *pull-up* a cualquier otro dispositivo esclavo que se conecte al bus I^{2}C de la Raspberry Pi.[4]

4.2. Paso 2: Lectura del sensor LM35

Antes de proceder, verifique conexiones con un multímetro en busca de corto circuitos. En particular verifique que exista una impedancia muy alta entre los pines VCC, GND y ADC_VREF del RP2040.

Para leer la temperatura con el RP2040 se necesitan convertir los valores discretos leídos por el ADC del microcontrolador en valores de temperatura. Esto se puede realizar mediante un simple análisis debido a la linearidad del LM35. Se tienen dos lecturas en el ADC: V_{OUT+} y V_{OUT-}, de las cuales la segunda es la referencia del LM35 y por lo tanto, la diferencia entre estos voltajes será proporcional a la temperatura en escala centígrada. Esto expresado matemáticamente es:

$$T[^oC] \propto V_{diff} = V_{OUT+} - V_{OUT-}$$

o bien

$$T[^oC] = k \times V_{diff} = k \times \left(V_{OUT+} - V_{OUT-}\right)$$

lo que implica que en $T = 0^oC; V_{OUT+} = V_{OUT-} \rightarrow V_{diff} = 0$

Es necesario entonces calcular la constante de proporcionalidad k. Sabemos que el ADC entregará lecturas de 0 a 4096 para los voltajes registradoes entre GND y ADC_VREF (0V y 2.72V respectivamente), además de que $1^oC = 0.01V$. Luego entonces

$$T[^oC] = V_{diff} \times \frac{2.72[V]}{4096 \times 0.01[\frac{V}{^oC}]}$$

$$T[^oC] = V_{diff} \times \frac{2.72}{40.96}[^oC]$$

o bien, generalizando para todo voltaje de referencia:

$$T[^oC] = V_{diff} \times \frac{V_{REF}}{40.96}[^oC]$$

Esta fórmula de conversión de unidades deberá programarse en el microcontrolador que adquiera los valores discretos de temperatura del sensor.

[4]Para más información sobre el papel de las resistencias de acoplamiento a positivo o *pull-up* en un bus I^{2}C se puede consultar http://dsscircuits.com/articles/effects-of-varying-i2c-pull-up-resistors

TIP: Calibración

El voltaje de referencia V_{REF} variará respecto a su valor teórico dependiendo de la tolerancia de las resistencias (típicamente $\pm 5\,\%$) y de las impedancias de los demás componentes conectados, incluyendo al RP2040 mismo.

Se recomienda usar siempre un multímetro para corroborar el voltaje entre V_{REF} y tierra y poder actualizar dicho valor en el código.[5]

El programa de ejemplo para el RP2040[6] se presenta a continuación:

Código ejemplo 1: `pico-code-adc.py`:23–38, `read_temp` function

```
def read_temp():
    '''
        Reads temperature in C from the ADC
    '''
    # The actual temperature
    vplus  = adcp.read_u16()
    # The reference temperature value, i.e. 0°C
    vminus = adcm.read_u16()
    # Calculate the difference. when V+ is smaller than V- we have negative temp
    vdiff  = vplus - vminus
    # Now, we need to convert values to the ADC resolution, AKA 2.72V/4096
    # We also know that 1°C = 0.01V so we can multiply by 2.72V / (0.01V/°C) = 272°C
    # to get °C instead of V. Analogously we can multiply VAREF by 100 but
    # since we will divide per 4096, it suffice with dividing by 40.96
    temp = vdiff * VAREF / 40.96
# end def
```

En ocasiones los valores pueden fluctuar ligeramente debido a ruido o variaciones de voltaje. Para evitar este tipo de imprecisiones es común utilizar técnicas de filtrado, y uno de los métodos más simples y comúnes es el promedio de varias lecturas consecutivas tal y como se muestra a continuación:

Código ejemplo 2: `pico-code-adc.py`:41–49, `read_avg_temp` function

```
def read_avg_temp(count=10):
    '''
        Gets the average of N temperature reads
    '''
    avgtemp = 0
    for i in range(count):
        avgtemp += read_temp()
    return avgtemp / count
# end def
```

Otro método mucho más eficaz y seguro es llevar un registro de las últimas N lecturas del sensor en un buffer circular y estimar el siguiente valor probable, descartando aquellas lecturas que estén fuera de rango posible, es decir cuando $\Delta t \geq \epsilon_0$.

4.3. Paso 3: Configuración de comunicaciones I²C

Primero ha de configurarse la Raspberry Pi para funcionar como dispositivo maestro o *master* en el bus I²C. Para esto, inicie la utilidad de configuración de la Raspberry Pi con el comando

```
# raspi-config
```

y seleccione la opción 5: Opciones de Interfaz (*Interfacing Options*) y active la opción `P5` para habilitar el I²C.

A continuación, verifique que el puerto I²C no se encuentre en la lista negra. Edite el archivo `/etc/modprobe.d/raspi-blacklist.conf` y revise que la línea `blacklist spi-bcm2708` esté comentada con #.

[5] El valor real de V_{REF} variará entre 2.4V y 3.1V con resistencias con tolerancia de 5 %, equivalente a una variación de 70°C.

[6] Para cargar el código de Python en el RP2040 utilice el editor Thonny. Para que los cambios sean persistentes, salve el archivo dentro del microcontrolador con el nombre `main.py`. Recuerde que no es posible programar el RP2040 via I²C.

Código ejemplo 3: /etc/modprobe.d/raspi-blacklist.conf

```
# blacklist spi and i2c by default (many users don't need them)
# blacklist i2c-bcm2708
```

Como paso siguiente, se habilita la carga del driver I²C. Esto se logra agregando la línea `i2c-dev` al final del archivo `/etc/modules` si esta no se encuentra ya allí.

Por último, se instalan los paquetes que permiten la comunicación mediante el bus I²C y se habilita al usuario predeterminado *pi* (o cualquier otro que se esté usando) para acceder al recurso.

```
# apt-get install i2c-tools python3-smbus
# adduser pi i2c
$ pip install smbus2
```

Reinicie la Raspberry Pi y pruebe la configuración ejecutando `i2cdetect -y 1` para buscar dispositivos conectados al bus I²C. Debería ver una salida como la siguiente:

```
$ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

4.4. Paso 4: Bitácora de temperatura via I²C

Con la Raspberry Pi configurada, basta con generar los dos programas para transferir las temperaturas registradas en el RP2040 a la Raspberry Pi que se encargará de almacenar esta información en un archivo o bitácora.

Primero, es necesario configurar al RP2040 como dispositivo esclavo e inicializar el bus I²C, tal como se muestra en los Códigos de Ejemplo 4 y 5.

Python es un lenguaje MUY lento que carece de tipos nativos primitivos y por ende hace un uso excesivo del *heap*, lo que hace imposible gestionar interrupciones de forma rápida y segura. Por este motivo las comunicaciones via I²C son **SÍNCRONAS** y requieren verificar el estado de los registros internos de forma contínua en un loop.

Además, MicroPython sólo ofrece soporte para utilizar los módulos I²C en configuración maestro. Luego entonces es necesario realizar la configuración y la lectura y escritura de datos a nivel registro, mecanismo que se conoce como envoltorio o *wrapping*. Dicho código se provee por conveniencia en el archivo `i2cslave.py`.

Código ejemplo 4: `pico-code-i2c.py:14-19` — Dirección asignada al dispositivo esclavo

```
1 from i2cslave import I2CSlave
2 VAREF          = 2.7273
3 I2C_SLAVE_ADDR = 0x0A
```

Código ejemplo 5: `pico-code-i2c.py:51-56` — Configuración del bus I²C y funciones de control

```
1 def setup():
2   global i2c, adcm, adcp
3   i2c = I2CSlave(address=I2C_SLAVE_ADDR)
4   adcm = machine.ADC(0)          # Init ADC0
5   adcp = machine.ADC(1)          # Init ADC1
6 # end def
```

El envío de datos se realiza byte por byte, por lo que es necesario convertir la medición de temperatura (*objeto tipo float*) en un arreglo de bytes que pueda ser transmitido. Esto se hace con la función `pack` de `ustruct`, librería dedicada a la conversión de objetos de python a arreglos de bytes que representen los tipos primitivos que operan en C Código de Ejemplo 6.

Código ejemplo 6: `pico-code-i2c.py:25-33` — Envío de datos

```
    # 1. Get temperature
    temperature = read_temp()
    # 2. Convert temperature from pyfloat to bytes
    data = ustruct.pack('<f', temperature)

    # 3. Check if Master requested data
    if i2c.waitForRdReq(timeout=0):
      # If so, send the temperature to Master
      i2c.write(data)
```

Del lado de la Raspberry Pi, primero ha inicializarse el bus I^2C mediante el uso de la librería `smbus`[7]y posteriormente se realizarán las lecturas en un poleo o bucle infinito, cada una de las cuales se irá almacenando en un archivo bitácora. La inicialización del bus requiere de una simple línea (véase Código de Ejemplo 7).

Código ejemplo 7: `raspberry-code-i2c.py:26` — Configuración del bus I^2C

```
import smbus2
import struct
# Initialize the I2C bus;
# RPI version 1 requires smbus.SMBus(0)
i2c = smbus2.SMBus(1)
```

La conversión de un arreglo de bytes a punto flotante en Python no es inmediata. Para esta opreación se utilizará la librería `struct` (véase Código de Ejemplo 7) que tomará los cuatro paquetes de 1 byte recibidos vía I^2C del RP2040 y los convertira en un *float*, como se muestra en el Código de Ejemplo 8. Nótese el símbolo $<$ (menor qué) a la izquierda del especificador de formato f, el cual se utiliza para definir el endianness de la transmisión de la información.

Código ejemplo 8: `raspberry-code-i2c.py:28-35` — Lectura de flotantes del bus I^2C

```
def readTemperature():
  try:
    msg = smbus2.i2c_msg.read(SLAVE_ADDR, 4)
    i2c.i2c_rdwr(msg)  # Performs write (read request)
    data = list(msg)   # Converts stream to list
    # list to array of bytes (required to decode)
    ba = bytearray()
    for c in data:
      ba.append(int(c))
    temp = struct.unpack('<f', ba)
    print('Received temp: {} = {}'.format(data, temp))
    return temp
  except:
```

El resto del programa es trivial, pues consiste sólo en la escritura del *timestamp UNIX* y el valor de temperatura registrado en un archivo de texto y la lectura de datos del RP2040 cada segundo.

Por conveniencia, los códigos completos de los programas de ejemplo se encuentran en los Apéndices A a C.

5. Experimentos

1. [6pt] Modifique el código de la subsección 4.4 para que la Raspberry Pi imprima en pantalla los valores de temperatura leídos.

2. [4pt] Modifique el código de la subsección 4.4 la Raspberry Pi grafique el histórico de temperaturas registradas, leyendo los valores almacenados e ingresados en la bitácora.

3. [+5pt] Con base en lo aprendido, modifique el código de la subsección 4.4 para que la Raspberry Pi sirva una página web donde se pueda observar la gráfica de temperatura (histórico) desde la bitácora con resolución de hasta 1 minuto.

[7]La implementación de la práctica utiliza `smbus2` que es una reimplementación codificada exclusivamente en Python de la librería `smbus` que es un *wrapper* de la *smbuslib* de C.

6. Referencias

Referencias

[1] *LM35 Precision Centigrade Temperature Sensors*. Texas Instruments, August 1999. Revised: December, 2017.

[2] *RP2040 Datasheet: A microcontroller by Raspberry Pi*. Raspberry Pi Ltd, March 2023. build-version: ae3b121-clean.

[3] I2C Info: A Two-wire Serial Protocol. I2c info – i2c bus, interface and protocol, 2020. https://i2c.info/, Last accessed on 2020-03-01.

[4] Charles Bell. Introducing the raspberry pi pico. In *Beginning MicroPython with the Raspberry Pi Pico: Build Electronics and IoT Projects*, pages 1–42. Springer, 2022.

A. Programa Ejemplo: `pico-code-adc.py`

src/pico–code–adc.py

```
from machine import ADC          # Board Analogic-to-Digital Converter
from utime import sleep_ms       # Delay function in milliseconds

VAREF = 2.72

def setup():
    '''
        Setup the Pico
    '''
    adcm = machine.ADC(0)         # Init ADC0
    adcp = machine.ADC(1)         # Init ADC1
# end def


def read_temp():
    '''
        Reads temperature in C from the ADC
    '''
    # The actual temperature
    vplus  = adcp.read_u16()
    # The reference temperature value, i.e. 0°C
    vminus = adcm.read_u16()
    # Calculate the difference. when V+ is smaller than V- we have negative temp
    vdiff  = vplus - vminus
    # Now, we need to convert values to the ADC resolution, AKA 2.72V/4096
    # We also know that 1°C = 0.01V so we can multiply by 2.72V / (0.01V/°C) = 272°C
    # to get °C instead of V. Analogously we can multiply VAREF by 100 but
    # since we will divide per 4096, it suffice with dividing by 40.96
    temp = vdiff * VAREF / 40.96
# end def


def read_avg_temp(count=10):
    '''
        Gets the average of N temperature reads
    '''
    avgtemp = 0
    for i in range(count):
        avgtemp += read_temp()
    return avgtemp / count
# end def


def main():
    while(True):                      # Repeat forever
        temp = read_avg_temp()        # Fetch temperature
        print(f'Temp: {temp:0.2f}°C') # Print temperature
        sleep_ms(1000)                # Wait for 1000ms
#end def

if __name__ == '__main__':
    main()
```

B. Programa Ejemplo: `raspberry-code-i2c.py`

src/raspberry–code–i2c.py

```python
import smbus2
import struct
import time

# RP2040 I2C device address
SLAVE_ADDR = 0x0A # I2C Address of RP2040

# Name of the file in which the log is kept
LOG_FILE = './temp.log'

# Initialize the I2C bus;
# RPI version 1 requires smbus.SMBus(0)
i2c = smbus2.SMBus(1)

def readTemperature():
  try:
    msg = smbus2.i2c_msg.read(SLAVE_ADDR, 4)
    i2c.i2c_rdwr(msg)  # Performs write (read request)
    data = list(msg)   # Converts stream to list
    # list to array of bytes (required to decode)
    ba = bytearray()
    for c in data:
      ba.append(int(c))
    temp = struct.unpack('<f', ba)
    print('Received temp: {} = {}'.format(data, temp))
    return temp
  except:
    return None

def log_temp(temperature):
  try:
    with open(LOG_FILE, 'w+') as fp:
      fp.write('{} {}°C\n'.format(
        time.time(),
        temperature
      ))
  except:
    return

def main():
  while True:
    try:
      cTemp = readTemperature()
      log_temp(cTemp)
      time.sleep(1)
    except KeyboardInterrupt:
      return

if __name__ == '__main__':
  main()
```

C. Programa Ejemplo: `pico-code-i2c.py`

src/pico–code–i2c.py

```
from i2cslave import I2CSlave
from utime import sleep_ms, sleep_us
import ustruct

VAREF          = 2.7273
I2C_SLAVE_ADDR = 0x0A


def main():
  setup()
  while True:
    # 1. Get temperature
    temperature = read_temp()
    # 2. Convert temperature from pyfloat to bytes
    data = ustruct.pack('<f', temperature)

    # 3. Check if Master requested data
    if i2c.waitForRdReq(timeout=0):
      # If so, send the temperature to Master
      i2c.write(data)
    # end if

    # 3. Check if Master sent data
    if i2c.waitForData(timeout=0):
      # If so, print it
      rcv = i2c.read()
      print( rcv.decode('utf-8') )
    # end if
# end def


def read_temp():
  # '''Reads temperature in C from the ADC'''
  return 25.0
# end def


def setup():
  global i2c, adcm, adcp
  i2c = I2CSlave(address=I2C_SLAVE_ADDR)
  adcm = machine.ADC(0)          # Init ADC0
  adcp = machine.ADC(1)          # Init ADC1
# end def


if __name__ == '__main__':
  main()
```

D. Programa Ejemplo: `i2cslave.py`

src/i2cslave.py

```
import machine
from utime import sleep_ms, sleep_us
class I2CSlave():
  def __init__(self, id=0, address=0x27, sda=None, scl=None):
    if id > 1: raise ValueError('Unsupported')
    elif id == 0:
      if sda is None: sda = 0
      if scl is None: scl = 1
      if sda not in [0, 4, 8, 12, 16, 20]:
        raise ValueError('Invalid pin number for sda')
      if scl not in [1, 5, 9, 13, 17, 21]:
        raise ValueError('Invalid pin number for scl')
    else:
      if sda is None: sda = 2
      if scl is None: scl = 3
      if sda not in [2, 6, 10, 14, 18, 26]:
        raise ValueError('Invalid pin number for sda')
      if scl not in [3, 7, 11, 15, 19, 27]:
        raise ValueError('Invalid pin number for scl')

    if address < 0 or address > 0x7f:
      raise ValueError('Address out of range (7bit addresses only)')

    self._sda  = sda
    self._scl  = scl
    self._addr = address
    self._base = __I2C0_BASE if id == 0 else __I2C1_BASE


    self.__setupPin(sda)
    self.__setupPin(scl)



    self.__regClr(__IC_ENABLE, 0x0001)

    self.__regClr(__IC_SAR, 0x01ff)
    self.__regSet(__IC_SAR, address)



    self.__regClr(__IC_CON, 0x0049)


    self.__regSet(__IC_ENABLE, 0x0001)

    self.__initialized = True


  def __setupPin(self, pin):


    gpioaddr = __IO_BANK0_BASE + 8 * pin + 4
    machine.mem32[ gpioaddr | __ATOM_CLR ] = 0x1f
    machine.mem32[ gpioaddr | __ATOM_SET ] = 0x03


  def __regClr(self, reg, mask):
    machine.mem32[self._base | __ATOM_CLR | reg] = mask


  def __regSet(self, reg, mask):
    machine.mem32[self._base | __ATOM_SET | reg] = mask

```

```
66
67    def __regRead(self, reg, andmask=0xffffffff):
68      return machine.mem32[self._base | __ATOM_RW | reg] & andmask
69
70    def __regWrite(self, reg, value):
71      machine.mem32[self._base | __ATOM_RW  | reg] = value
72
73
74    def __regXor(self, reg, mask):
75      machine.mem32[self._base | __ATOM_XOR | reg] = mask
76
77
78    @property
79    def id(self):
80      return 0 if self._base == __I2C0_BASE else 1
81
82
83    @property
84    def sda(self):
85      return self._sda
86
87
88    @property
89    def scl(self):
90      return self._scl
91
92
93    @property
94    def address(self):
95      return self._addr
96
97    def idle(self):
98      return not self.__regRead(__IC_STATUS, 0x01)
99
100
101   def rxBufferCount(self):
102     return self.__regRead(__IC_RXFLR, 0x1f)
103
104
105   def rxBufferEmpty(self):
106     return not self.__regRead(__IC_STATUS, 0x08)
107
108
109   def rxBufferFull(self):
110     return self.__regRead(__IC_STATUS, 0x10)
111
112
113   def txBufferCount(self):
114     return self.__regRead(__IC_TXFLR, 0x10)
115
116
117   def txBufferEmpty(self):
118     return not self.__regRead(__IC_STATUS, 0x04)
119
120
121   def txBufferFull(self):
122     return not self.__regRead(__IC_STATUS, 0x02)
123
124
125   def read(self):
126
127
128     if not self.__initialized: raise IOError('Uninitialized')
129
130     while not self.__regRead(__IC_STATUS, 0x08) and self.__regRead(__IC_RXFLR, 0x1f) < 1:
131       sleep_us(10)
132
133     bytecount = self.__regRead(__IC_RXFLR, 0x1f)
```

```
134     ba = bytearray(bytecount)
135     for i in range(bytecount):
136       ba[i] = self.__regRead(__IC_DATA_CMD, 0xff)
137     return ba
138 
139 
140   def readByte(self):
141 
142 
143     if not self.__initialized: raise IOError('Uninitialized')
144 
145     while not self.__regRead(__IC_STATUS, 0x08):
146       sleep_us(10)
147 
148     return self.__regRead(__IC_DATA_CMD, 0xff)
149 
150 
151   def write(self, ba):
152 
153 
154 
155 
156 
157     if not self.__initialized: raise IOError('Uninitialized')
158     if not isinstance(ba, (bytes, bytearray)): raise ValueError('An array of bytes is
      required')
159     for b in ba:
160       self.writeByte(b)
161 
162 
163   def writeByte(self, b):
164 
165 
166 
167 
168 
169     if not self.__initialized: raise IOError('Uninitialized')
170 
171     while not self.__regRead(__IC_RAW_INTR_STAT, 0x20):
172       sleep_us(10)
173 
174     self.__regClr(__IC_CLR_TX_ABRT, 0x01)
175 
176     self.__regRead(__IC_CLR_RD_REQ)
177 
178     while not self.__regRead(__IC_STATUS, 0x02):
179       sleep_us(10)
180     self.__regWrite(__IC_DATA_CMD, b & 0xff)
181 
182 
183   def waitForData(self, timeout=-1):
184 
185 
186 
187     if not self.__initialized: raise IOError('Uninitialized')
188     while self.rxBufferEmpty() and (timeout > 0):
189       timeout-= 1
190       sleep_ms(1)
191     return not self.rxBufferEmpty()
192 
193 
194   def waitForRdReq(self, timeout=-1):
195 
196 
197 
198     if not self.__initialized: raise IOError('Uninitialized')
199     while (self.__regRead(__IC_RAW_INTR_STAT, 0x20) != 0) and (timeout > 0):
200       timeout-= 1
```

```
201         sleep_ms(1)
202       return self.__regRead(__IC_RAW_INTR_STAT, 0x20)
203
204
```