Práctica 7:
Modulación de potencia de una lámpara incandecente usando el RP2040 y la Raspberry Pi

Fundamentos de Sistemas Embebidos

Autor: José Mauricio Matamoros de Maria y Campos

1. Objetivo

El alumno aprenderá a modular la potencia de una carga resistiva de alta potencia opto-acoplada a un microcontrolador por medio de un detector de cruce por cero y un TRIAC.

2. Introducción

La presente práctica resume los pasos a seguir para modular la cantidad de corriente que pasa por un foco incandescente con un microcontrolador. En particular, se interesa en el uso de un circuito de detección de cruce por cero para conmutar un triac acoplado a un microcontrolador RP2040 utilizando MicroPython. Los datos registrados serán posteriormente enviados vía I^2C a una Raspberry Pi para controlar la intencidad del foco.

2.1. El puente rectificador

Un puente rectificador o rectificador de onda completa es un arreglo de 4 diodos que permiten el paso de la corriente en un sólo sentido, invirtiendo así la parte negativa de una señal de AC respecto a su voltaje de referencia. Los puentes rectificadores son un componente fundamental en los transformadores de corriente de AC a DC.

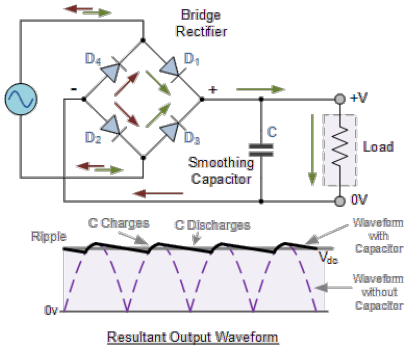


Figura 1: Rectificador de onda completa[1]

El circuito funciona de la siguiente manera. En la primera parte del ciclo, cuando el voltaje comienza a aumentar, la corriente fluye de la parte norte del puente (arriba) a través de D_1 hacia la carga, y luego de regreso por D_2 hacia el neutro (véase Figura 1). En esta primera etapa, D_3 y D_4 actuan como barreras evitando que la corriente fluya directamente de la fase al neutro (corto circuito). Tras pasar por la carga, el voltaje en en el ánodo de D_4 ha caído y es menor que en el cátodo, por lo que no habrá flujo de corriente en esta dirección, pero aún es positivo respecto al neutro ($V_N = 0$), por lo que la corriente tendrpa que pasar por D_1. De forma análoga, el voltaje en el cátodo de D_3 es menor que en el ánodo, por lo que tampoco habrá flujo en esta dirección.

En la segunda parte del ciclo, el voltaje de fase disminuye respecto al neutro, por lo que la corriente fluye de la parte sur del puente (abajo) a través de D_3 hacia la carga, y luego de regreso por D_4 hacia la fase (véase Figura 1). En esta segunda etapa, D_1 y D_2 actuan como barreras evitando que la corriente fluya directamente del neutro a la fase (corto circuito). Tras pasar por la carga, el voltaje en en el ánodo de D_2 ha caído y es menor que en el cátodo, por lo que no habrá flujo de corriente en esta dirección, pero aún es positivo respecto a la fase ($V_N = 0$), por lo que la corriente tendrpa que pasar por D_4. De forma análoga, el voltaje en el cátodo de D_1 es menor que en el ánodo, por lo que tampoco habrá flujo en esta dirección.

[1] Fuente de imagen: https://lasopaeden528.weebly.com/bridge-rectifier-calculator.html

2.2. Optoacopladores

Un optoacoplador o optoaislador es un circuito integrado que permite aislar mecánicamente dos circuitos, por lo que se usa comunmente para separar la lógica de control de los circuitos de potencia. El principio básico de un optoacoplador, como su nombre lo indica, es utilizar transductores ópticos para la transmisión de señales eléctricas. Así, en un lado del optoacoplador se tendrá siempre un diodo LED y en el otro extremo un fotoreceptor, que puede ser un foto SRC, un foto dárlington, un foto TRIAC o un fototransistor, siendo este último el más común.

En un optoacoplador, el diodo LED emite una cantidad de luz directamente proporcional a la corriente que circula por éste y, al incidir ésta en el fotoreceptor, el estímulo luminoso activa el paso de corriente a través de éste. Por ejemplo, en el caso de un fototransistor, al incidir la luz en la juntura de la base ésta se ioniza, generando un puente de iones que permite el flujo entre los extremos del transistor. Además, la mayoría de los optoacopladores tienen un pin conectado directamente a la base que sirve para ajustar la sensibilidad de la misma mediante la inyección de un voltaje pequeño.

Los fototransistores foto-dárlingtons se utilizan principalmente en circuitos DC, mientras que los foto SCR y los foto TRIACs permiten controlar los circuitos de AC. Existen muchos otros tipos de combinaciones de fuente-sensor tales como LED-fotodiodo, LED-LÁSER, pares de lámpara-fotorresistencia, optoacopladores reflectantes y ranurados.

Por ejemplo, el integrado 4N25 puede usarse para monitorear el voltaje de la línea de tensión doméstica con un integrado. Según su hoja de especificaciones [1], la entrada del 4N25 acepta hasta 60mA, permite un flujo por el colector de hasta 50mA, y aisla hasta $5000\mathrm{V}_{\mathrm{RMS}}$. Si se toma como entrada un voltaje de línea rectificado de $127\mathrm{V}_{\mathrm{RMS}}$ y se limita la corriente a la corriente de prueba de 50mA indicada en la hoja de especificaciones [1], el 4N25 tendrá que acoplarse con una resistencia de al menos 2K7Ω aproximada mediante la fórmula:

$$R = \frac{V}{I} = \frac{127V_{\mathrm{RMS}}}{0.050A} = 2540\Omega$$

Sin embargo, como $P = I^2R$ una resistencia de 2k7Ω disiparía $(47mA)^2 \times 2.7k\Omega = 5.96\mathrm{Watt}$, que es un tremendo desperdicio de energía disipada como calor.

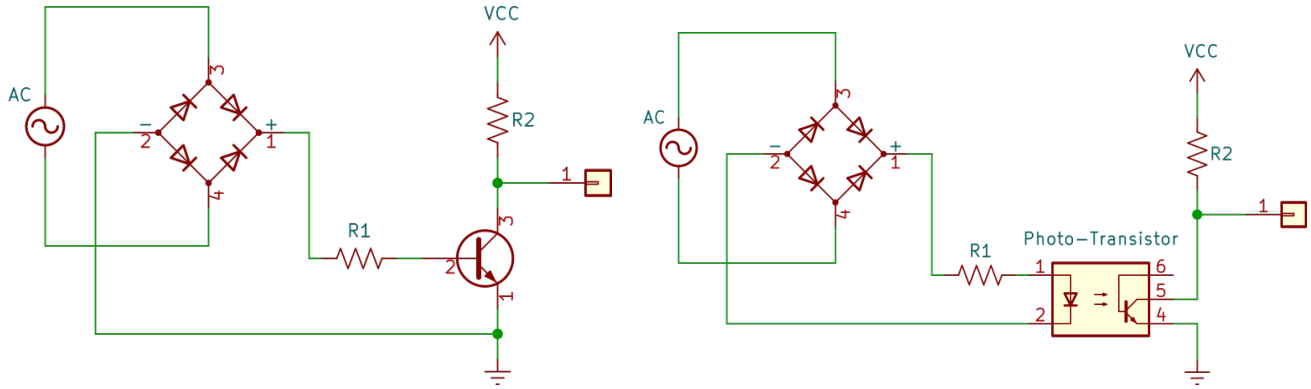
En este caso conviene reducir mucho más la corriente que circula por el 4N25. Supóngase que el propósito del 4N25 fuera el de activar una interrupción en un microcontrolador. Normalmente las entradas digitales de los microcontroladores requieren de unos cuantos microamperios para activarse, por lo que pueden elegirse $10\mu A$ como un valor conservador seguro. Por otro lado, la hoja de especificaciones nos indica que la corriente que circula por el transistor será el 50 % de la corriente que pase por el fotodiodo [1]; por lo que la resistencia elegida tendrá que limitar la corriente a por lo menos $20\mu A$, en este caso:

$$R = \frac{V}{I} = \frac{127V_{\mathrm{RMS}}}{20 \times 10^{-6}A} = 6350000\Omega \approx 5\mathrm{M}6\Omega$$

Otro enfoque es considerar las resistencias comerciales más económicas en el mercado, es decir las de $^1/_4$Watt. Como $P = \frac{V^2}{R} = 0.25\mathrm{Watt}$ y $V = 127V$ entonces $R_{\mathrm{Max}} = \frac{(127V)^2}{0.25\mathrm{Watt}} = \frac{16129V^2}{0.25\mathrm{Watt}} = 64516\Omega \approx 68\mathrm{K}\Omega$ cualquier resistencia de 68KΩ o mayor será suficientemente grande para optoacoplar un microcontrolador a una línea de tensión de 127VAC con un 4N25 sin disipar mucho calor, proporcionando un flujo por el fototransistor de 0.9mA; más que suficiente para drenar un pin digital acoplado a V_{CC} con una resistencia de *pull-up* de 10KΩ.

2.3. Detector de cruce por cero

Un circuito detector de cruce por cero es un circuito electrónico diseñado para detectar cuando una señal senoidal pasa por cero. Estos circuitos se usan comunmente en electrónica de potencia tanto para detectar la frecuencia de la línea y hacer cálculo de fases. Además, al reducirse la diferencia de potencial a cero entre la fase y el neutro, la corriente instantánea también es cero, haciendo de éste el momento ideal para cortar la alimentación sin dañar las cargas.



(a) Detector de cruce por cero simple

(b) Detector de cruce por cero optoacoplado

Figura 2: Detectores de cruce por cero

La forma más sencilla de alambrar un circuito detector de cruce por cero es mediante un puente rectificador, dos resistencias y un transistor tipo NPN en modo interruptor (véase Figura 2a). El puente rectificador se encarga de invertir la parte negativa de la señal de AC, evitando así corrientes inversas que el transistor es incapaz de manejar. Cuando hay voltaje en la línea, éste habilita la base cerrando el circuito del transistor y conectando el pin de sensado a tierra (la resistencia de base evita el corto circuito y ajusta el umbral de sensibilidad). Tan pronto como la diferencia de potencial entre la línea y el neutro cae a cero (o suficientemente bajo como la resistencia de base permita) el circuito se abre y en el pin de sensado se registra Vcc.

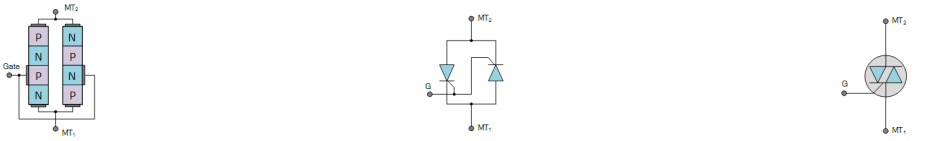
Para calcular R_1 es necesario tomar en cuenta las características eléctricas del transistor y los voltajes de pico de la línea. Se sabe que $V_{\text{pico}} = V_{\text{RMS}}\sqrt{2}$, por lo que usando la ley de ohm se tiene:

$$R_1 = \frac{V_{\text{RMS}}\sqrt{2}}{i_{\text{transistor}}} \approx \frac{1.4142 V_{\text{RMS}}}{i_{\text{transistor}}} \tag{1}$$

Sin embargo, el voltaje de la línea rara vez viene rectificado y un transitorio de corriente derivado de una descarga inductiva (arranque de refrigerador o microondas) puede incluso duplicar el voltaje de la línea, quemando no sólo el transistor sino el microprocesador. Es por esto que es muy aconsejable utilizar un optoacoplador en lugar de un simple transistor, tal como muestra la figura Figura 2b. Los principios de operación son los mismos.

2.4. TRIACs

Un TRIAC o triodo interruptor para corriente alterna (*Triode AC Switch*) es un integrado de estado sólido compuesto por dos tristores conectados en paralelo inverso (véase Figura 3b) que permite conmutar la corriente que pasa por un circuito de AC a alta frecuencia de manera similar a como operan los transistores bipolares y FETs en DC. Es decir, un TRIAC es un interruptor de estado sólido que puede operar a gran velocidad que, a diferencia de los relés, no existe la posibilidad de que un arco eléctrico funda los metales y el dispositivo se quede en encendido permanente, sino que al quemarse un TRIAC siempre abre el circuito. Por otro lado, basta una corriente muy pequeña entre el gate (G) y cualquiera de las terminales (MT_1 y MT_2) para encender al TRIAC, lo que lo convierte en el aliado ideal para controlar dispositivos de alta potencia con un microcontrolador.



(a) Construcción física

(b) Equivalente en tristores

(c) Símbolo

Figura 3: El Triac[2]

[2]Fuente de imagen: https://www.electronics-tutorials.ws/power/triac.html

Como siempre, al utilizar un TRIAC es deseable aislar la parte de corriente directa del circuito de la parte de corriente alterna, es decir, el TRIAC deberá estar aislado pero acoplado al circuito DC. Esto normalmente se realiza mediante el uso de optoacopladores tipo MOC, tal como se ilustra en la Figura 4.

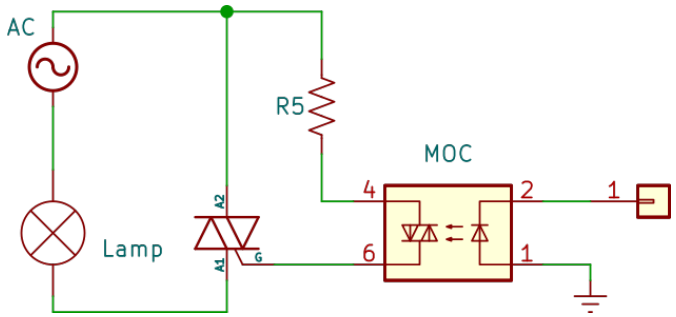


Figura 4: Triac optoacoplado

Una peculiaridad de los TRIAC es que el tiempo de encendido no es proporcional al tiempo que se inyecta una corriente por el *gate*, sino que una vez recibido el pulso de arranque el TRIAC permanecerá habilitado hasta el siguiente cruce por cero de la onda senoidal de corriente alterna. Es decir que, diferencia de los transistores de juntura bipolar (BJT) o de efecto de campo (FET), un TRIAC no puede usarse para modular corriente directa con un PWM. Por este motivo es necesario utilizar el **complemento** del ángulo de fase al modular la potencia de un dispositivo usando un TRIAC.

Por ejemplo, supóngase que se desea obtener el 50 % de potencia[3] de una resistencia, considerando que una resistencia tiene una respuesta lineal (el voltaje y la corriente están en fase) y que la senoidal de la línea de AC es una simétrica.

$$
\begin{aligned}
\cancel{V_P} &= \cancel{V_P} \cdot \sin(2\pi \cdot f \cdot t) \\
1 &= \sin(2\pi \cdot f \cdot t) \\
\arcsin(1) &= \cancel{\arcsin}(\cancel{\sin}(2\pi \cdot f \cdot t)) \\
\frac{\pi}{2} &= 2\pi \cdot f \cdot t \\
t &= \frac{\frac{\cancel{\pi}}{2}}{\cancel{2\pi} \cdot f} \\
t &= \frac{1}{4f}
\end{aligned}
$$

Si la frecuencia de línea fuera de 50Hz, el tiempo de disparo $t_{50\,\%}$ después del cruce por cero sería:

$$
\begin{aligned}
t_{50\,\%} &= \frac{1}{4 \times 50\text{Hz}} \\
&= \frac{1}{200\frac{1}{s}} \\
&= 0.005s \\
&= 5ms
\end{aligned}
$$

De acuerdo con lo anterior, el triac deberá encenderse en $t = 5ms$ y así permanecerá encendido hasta que el voltaje de la línea pase por cero y se invierta, es decir, la mitad del ciclo. Ni siquiera es necesario mantener la señal de encendido del TRIAC durante este tiempo. Para encender el triac basta con un breve pulso normalmente despreciable (ej. $10\mu s$).

[3] Como $P = VI$ y $V = RI$ entonces $P = \frac{V^2}{R}$ donde V es el voltaje promedio o RMS suministrado. Aquí es necesario hacer una distinción entre el voltaje RMS nominal de línea V_L y el voltaje RMS de la onda recortada por del TRIAC V_α, pues la potencia

2.5. Modulación de potencia de carga resistiva en AC

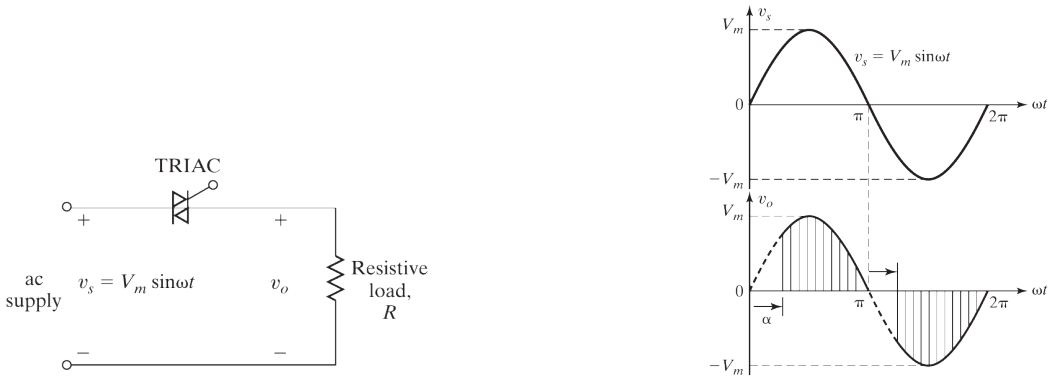
A diferencia de un circuito de DC, la modulación de la potencia de una carga en un circuito de AC de una fase incolucra cuatro parámetros: i) el voltaje en la carga, ii) la corriente que circula por la carga, iii) la impedancia de la carga y iv) el ángulo de fase. O, matemáticamente hablando:

$$p(t) = vi \tag{4}$$
$$= V_m \sin(\omega t + \theta_v)\, I_m \sin(\omega t + \theta_i) \tag{5}$$

donde:

- ω la velocidad angular tal que $\omega = 2\pi f$
- V_m el voltaje máximo
- I_m la corriente máxima
- θ_v el ángulo de fase del voltaje
- θ_i el ángulo de fase de la corriente

De estos cuatro parámetros, en un circuito puramente resistivo la impedancia R es constante, la corriente $i(\omega t + \theta_i)$ es proporcional al voltaje a la entrada de la carga y a la resistencia de la misma, y el voltaje $v(\omega t + \theta_v)$ oscila en el rango $[-V_m, V_m]$, con V_m el voltaje de pico. No obstante, se puede modificar voltaje promedio en la carga al variar el ángulo de fase y así controlar la potencia. Esta técnica es de hecho el principio fundamental de los circuitos convertidores de AC-AC a base de TRIAC como el que se muestra en la Figura 5.



(a) Circuito

(b) Señal voltaica

Figura 5: Convertidor AC-AC de una fase.[4]

Nótese que los ángulos de defasamiento de voltaje θ_v y corriente θ_i han desaparecido. Esto se debe a que en un circuito de AC de una sola fase $\theta_v = 0$ por ser la única fase y por ende la referencia. Además, en un circuito puramente resistivo las señales de voltaje y corriente están acopladas, por lo que $\theta_i = \theta_v$ tal como se muestra en la Figura 6.

suministrada se corresponderá con este último de la forma:

$$V_\alpha = V_L \sqrt{\frac{1}{\pi}\left(\pi - \alpha + \frac{\sin(2\alpha)}{2}\right)} \text{ ; donde } \alpha = \omega t \text{ y } \omega = 2\pi f \tag{2}$$

Cuando lo que interesa es un percentil de la potencia suministrada se tiene que $P_{\text{MAX}} = P_L = P_{\alpha=0}$, por lo que se puede utilizar el cociente $\frac{P_\alpha}{P_L} = \frac{V_\alpha^2 \cancel{R}}{V_L^2 \cancel{R}}$ que, despejando en la Ecuación (2) produce:

$$P\,\% = \left(1 - \frac{\alpha}{\pi} + \frac{\sin(2\alpha)}{2\pi}\right) \times 100 \tag{3}$$

Que, a una frecuencia de 50Hz y con $t = 5ms$ ($\alpha = 0.5\pi$) reporta un 50 % de potencia.

Es importante remarcar que la Ecuación (3) es válida sólo si la impedancia de la carga es invariante en el tiempo ($\frac{dR}{dt} = 0$). Si este no fuere el caso (como por ejemplo con una lámpara incandescente) la potencia no variará de forma cuadrática proporcional con el voltaje.

[4]Fuente de la imagen: Rashid [2, pp. 33].

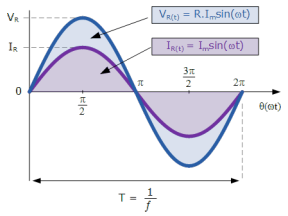


Figura 6: Voltaje y corriente en un circuito resistivo monofase.[5]

En la mayoría de los casos lo que interesa no es el cálculo de la potencia neta, sino controlar el factor de potencia, el decir, el porcentaje de la potencia máxima que la carga está entregando. Al estar sincronizadas la corriente y el voltaje por ser un circuito resistivo y no existir más que una fase, el cálculo de la potencia se simplifica y se convierte en el cociente del voltaje promedio aplicado a la carga respecto al voltaje RMS de la línea. Es decir

$$P_f = \frac{V_o}{V_{RMS}} \tag{6}$$

y dado que V_{RMS} es fijo, lo que interesa calcular es el voltaje aplicado a la carga V_o que se calcula como [2, 579–583]:[6]

$$\begin{aligned} V_o^2 =& \frac{2}{2\pi}\int_{\alpha}^{\pi} 2V_{RMS}^2 \sin^2(\omega t)\, d(\omega t) && (7)\\ =& \frac{4V_{RMS}^2}{4\pi}\int_{\alpha}^{\pi} \left(1-\cos(2\omega t)\right) d(\omega t) && (8)\\ =& \frac{V_{RMS}^2}{\pi}\left(\pi - \alpha + \frac{\sin(2\alpha)}{2}\right) && (9) \end{aligned}$$

Por lo tanto

$$\begin{aligned} V_o =& \left[\frac{V_{RMS}^2}{\pi}\left(\pi - \alpha + \frac{\sin(2\alpha)}{2}\right)\right]^{\frac{1}{2}} && (10)\\ =& V_{RMS}\sqrt{\frac{1}{\pi}\left(\pi - \alpha + \frac{\sin(2\alpha)}{2}\right)} && (11) \end{aligned}$$

Ahora, supóngase que se desea modular la potencia de un foco incandescente de $60W$ conectado a una línea estándar de $V_{RMS} = 120V$, 60Hz. Como $P = \frac{V^2}{R}$ se puede calcular tanto la resistencia del foco como la corriente a fin de elegir el TRIAC adecuado:

$$\begin{aligned} R =& \frac{V^2}{P}\\ =& \frac{(120V)^2}{60W}\\ =& \frac{14400V^2}{60W}\\ =& 240\Omega \end{aligned}$$

y como $I = \frac{V}{R}$

$$\begin{aligned} I_{RMS} &= \frac{120V}{240\Omega} = 0.5A\\ I_m &= \sqrt{2}\times I_{RMS} = \sqrt{2}\times 0.5A\\ &= 0.7A \end{aligned}$$

[6] El valor medio o efectivo de cualquier función $f(\omega t)$ con periodo de $2\pi rad$ está determinado por $F = \sqrt{\frac{1}{2\pi}\int_0^{2\pi} f^2(\omega t)\, d\omega t}$

Ahora bien, si se usa un ángulo de disparo en el TRIAC de $\alpha = \frac{\pi}{2}$ la potencia de el foco con base en las Ecuaciones (6) y (11) será:

$$\begin{aligned}
P_f &= \frac{V_o}{V_{RMS}} \\
&= \frac{\cancel{V_{RMS}} \left[\frac{1}{\pi}\left(\pi - \alpha + \frac{\sin(2\alpha)}{2}\right)\right]^{\frac{1}{2}}}{\cancel{V_{RMS}}} \\
&= \left[\frac{1}{\pi}\left(\pi - \frac{\pi}{2} + \frac{\sin\left(\frac{2\pi}{2}\right)}{2}\right)\right]^{\frac{1}{2}} \\
&= \left[\frac{1}{\pi}\left(\frac{\pi}{2} + \frac{\sin(\pi)}{2}^{0}\right)\right]^{\frac{1}{2}} \\
&= \left(\frac{1}{\cancel{\pi}} \cdot \frac{\cancel{\pi}}{2}\right)^{\frac{1}{2}} \\
&= \sqrt{\frac{1}{2}} \\
&= 0.707 \\
&= 70.7\,\%
\end{aligned}$$

De este desarrollo se concluye, además, que el factor de potencia no depende de los valores de voltaje, sino sólo del ángulo de disparo del TRIAC α.

Como $\alpha = \omega t$, para conocer el tiempo de disparo τ basta con sustituir $t = \tau$ y despejar. Así:

$$\begin{aligned}
\tau =& \frac{\alpha}{\omega} \\
=& \frac{\alpha}{2\pi f} \\
=& \left.\frac{\frac{\pi}{2}}{2\pi f}\right|_{f=60\text{Hz}} \\
=& \frac{1}{4 \times 60\text{Hz}} = \frac{1}{240\text{Hz}} \\
=& 0.004166\bar{6}\text{s} \\
\approx& 4.2\text{ms}
\end{aligned}$$

Un problema importante a considerar es que la ecuación $P_f = \left[\frac{1}{\pi}\left(\pi - \alpha + \frac{\sin(2\alpha)}{2}\right)\right]^{\frac{1}{2}}$ no es biyectiva (tiene una componente periódica senoidal) y por lo tanto no es posible calcular su inversa de forma analítica. En otras palabras, no es posible obtener una expresión algebraica para calcular $\alpha\,(P_f)$, y por tampoco es posible inferir $\tau\,(P_f)$.

Existen varias soluciones alterna a este problema. Por ejemplo, pueden utilizarse varios métodos de aproximación numérica para cada uno de los segmentos de la curva y utilizar el más adecuado para en cada uno de los rangos de interés. Otros métodos que requieren un número mucho menor de cálculos hacen uso de una tabla de valores discretos (por ejemplo incrementos de 1 % en el factor de potencia) e interpolan linealmente entre estos, dejando el error como una perturbación a corregir por el controlador (véase Tabla 1).

Tabla 1: Relación entre factor de potencia y tiempo de disparo de TRIAC
Tabla para interpolación con incrementos de 5 % y alimentación de AC a 60Hz

Factor de potencia [%]	**Tiempo de disparo [ms]**	**Factor de potencia [%]**	**Tiempo de disparo [ms]**	**Factor de potencia [%]**	**Tiempo de disparo [ms]**
100	0.000	65	4.464	30	6.232
95	2.060	60	4.734	25	6.487
90	2.696	55	4.993	20	6.750
85	3.157	50	5.245	15	7.030
80	3.538	45	5.493	10	7.334
75	3.874	40	5.738	5	7.688
70	4.179	35	5.984	0	8.203

2.6. Bus I²C

I²C es un protocolo serial inventado por Phillips y diseñado para conectar dispositivos de baja velocidad mediante interfaces de dos hilos (Figura 7). 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].

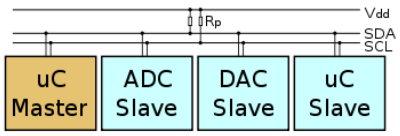


Figura 7: Bus I²C

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 (ej. Raspnerry Pi Pico) con MicroPython precargado.
- 1 TRIAC BT138 o BT139
- 4 diodos 1N4007 o puente rectificador equivalente
- 1 optoacoplador MOC 3021
- 1 optoacoplador 4N25
- 1 foco incandescente (NO AHORRADOR NI LED)
- 1 resistencia de 68kΩ, ¼Watt
- 1 resistencia de 10kΩ, ¼Watt
- 2 resistencia de 4k7Ω, ¼Watt
- 1 resistencia de 1kΩ, 1Watt
- 2 resistencia de 470Ω, ¼Watt
- 2 resistencia de 330Ω, ¼Watt
- 1 LED de 5mm
- 1 LED ultrabrillante de 5mm
- 1 protoboard o circuito impreso equivalente
- 1 fuente de alimentación regulada a 5V y al menos 2 amperios de salida
- Cables y conectores varios

4. Instrucciones

1. Alambre el circuito mostrado en las Figuras 8 y 9.
2. Realice los programas de la Subsección 4.3
3. Analice los programas de la subsección 4.3 y realice los experimentos propuestos en la sección 5.

4.1. Paso 1: Alambrado

El proceso de alambrado de esta práctica considera dos circuitos. El primer circuito (véase Figura 8) opera con corriente alterna e integra un detector de cruce por cero y un convertidor AC-AC con base en un TRIAC. Ambos subcircuitos cuentan con optoacopladores que servirán como interfaz para una conexión segura al circuito de DC.

El segundo circuito (véase Figura 9) está encargado de detectar el cruce por cero y enviar la señal de activación al TRIAC en el momento oportuno de acuerdo con la potencia requerida por el usuario (el brillo del foco) mediante una interfaz gráfica.

Advertencia

Asegúrese de que todos los cables para el circuito AC están perfectamente aislados.
Las puntas expuestas son un riesgo de electrocución y quemarán su arduino y su Pi con un sólo roce.

Utilice cinta aislante.

Para este fin, se alambra la señal del subcircuito detector de cruce por cero a un pin digital del RP2040 controlado por la PIO que iniciará una cuenta regresiva y enviará la señal de activación al TRIAC una vez transcurrido el tiempo de activación, obteniéndose así la potencia deseada. Asimismo, el RP2040 recibirá de la Raspberry Pi via I^2C la potencia solicitada por el usuario mediante una interfaz web.

Cabe mencionar que, al ser un circuito completamente digital, a la GPIO de la Raspberry Pi podría configurársele un pin en modo interrupción para recibir la señal del detector de cruce por cero y otro para el envío de la señal de activación del TRIAC. Sin embargo, el paquete `RPi.GPIO` para el control de la GPIO con Python no soporta el uso de interrupciones de timer en hardware, por lo que no es posible garantizar que la señal de activación del TRIAC será enviada sin retrasos, y mucho menos que el ciclo de trabajo será periódico y preciso. Es por este motivo que se utiliza un RP2040 como auxiliar.

4.1.1. Circuito de potencia en AC

Alambre primero el circuito de corriente alterna de la Figura 8 tras verificar los valores de las resistencias de los optoacopladores. Considere que si la resistencia de gatillo es muy grande (ej. 10MΩ), el optoacoplador no recibirá suficente corriente y no encenderá lo suficiente como para disparar el fotosensor. Por otro lado, si la resistencia es demasiado pequeña el optoacoplador se quemará irremediablemente.

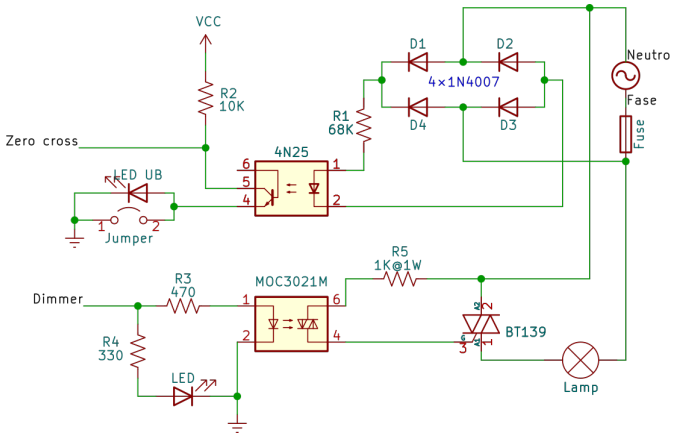


Figura 8: Circuito de potencia en AC

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

Pin Raspberry		Conexión			Pin RP2040	
3	(GPIO2)	Raspberry Pi SDA	→	RP2040 SDA	GP0/GP20	1/26
5	(GPIO3)	Raspberry Pi SCL	→	RP2040 SCL	GP1/GP21	2/27
6	(GND)	Raspberry Pi GND	→	RP2040 GND	GND	3/28

Tras alambrar el circuito, es una buena idea probar el detector de cruce por cero con un osciloscopio, o al menos con un led ultrabrillante (LED UB), que deberá encender tenuemente. De igual manera, conviene probar el encendido del foco inyectando 5V al MOC que acopla al TRIAC.

Importante

Asegúrese de verificar con un multímetro que el circuito de AC está debidamente aislado y que no se tienen valores mayores a 5V en el segmento de DC. De otro modo podría quemar su RP2040 y su Raspberry Pi.

Continúe el alambrado del circuito.

4.1.2. Circuito de control en DC

Alambre el circuito de corriente directa de la Figura 9 tras verificar la tensión de las señales optoacopladas conectando el bus I^{2}C entre la Raspberry Pi y el RP2040 como ilustran la Tabla 2 y la Figura 9. Hay tutoriales que sugieren utilizar un convertidor de niveles de voltaje cuando se conecta una Raspberry Pi a un RP2040 mediante I^{2}C. Esto **NO** es necesario si la Raspberry Pi está configurada como dispositivo maestro o *master* y el RP2040 como dispositivo esclavo o *slave*.

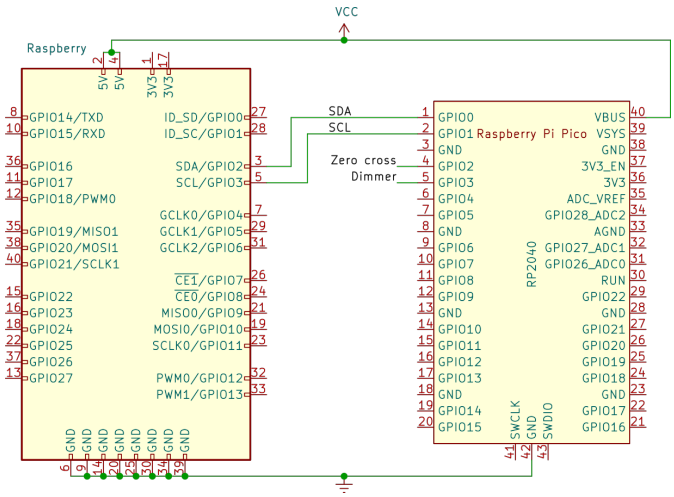


Figura 9: Circuito de control en DC

Esto es posible debido a que el RP2040 no tiene 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.[7]

Un estudiante avispado habrá podido observar que en la Tabla 2 aparecen dos posibles pines para su uso como SDA, SCL y GND en el RP2040. De acuerdo con la hoja de especificaciones del fabricante, es posible utilizar

[7]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

cualquier pin entre el GP0 y el GP27 con excepción del GP22 para comunicaciones I^2C, quedando repartidos los pines entre los periféricos I^2C0 e I^2C1 de forma alternada. Es decir, 2^{2n} y $2^{2n}+1$ para el I^2C0 y 2^{2n+1} y $2^{2n+1}+1$ para el I^2C1.

A continuación pruebe el alambrado del circuito de DC con el programa de prueba del Apéndice A. El programa es muy simple, pues sólo configura interrupciones e imprime el conteo de estas. Para activarlo inyecte repetidamente 5V con una resistenca de 1kΩ al pin de detección de cruce por cero. Deberá observar en la pantalla la frecuencia de activación del pin en Hz.

Al terminar el alambrado debería tener completo el citcuito de la Figura 10

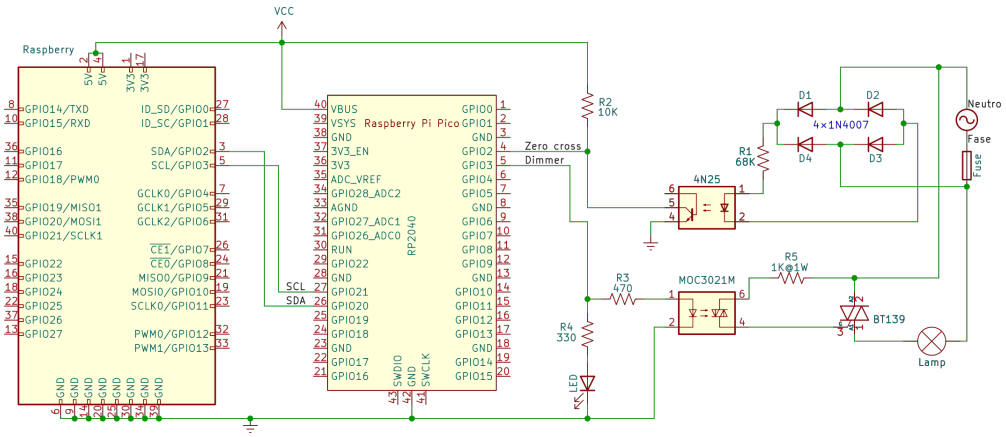


Figura 10: Diagrama de circuito alambrado completo

Una vez alambrado, pruebe su circuito con el programa de prueba del Apéndice D. Dicho programa utiliza la máquina de estados de la PIO (***P**rogrammable **I**nput **O**utput*) del RP2040. Este periférico permite programar los pines del RP2040 para descargar al procesador de las engorrosas tareas de transmisión y recepción de datos y que éste pueda enfocarse en realizar cómputo.

El uso de una PIO es necesario por un motivo fundamental: al ser un lenguaje interpretado y orientado a objetos, Python hace un uso extensivo del *heap* y de la pila cada vez que se llama a una función, reservando memoria de manera dinámica para todos los tipos, incluyendo enteros y flotantes. La reserva de memoria dinámica es una operación de tipo $O(n)$ respecto al tamaño de los objetos a reservar en la función, por lo que una simple llamada puede requerir de cientos o incluso miles de ciclos de reloj. Además, como Python hace uso de la pila,[8] lo que está prohibido al atender interrupciones, el intérprete encola las interrupciones para su ejecución concurrente a posteriori en lugar de atenderlas inmediatamente. Por otro lado, el uso de un PWM no es posible debido principalmente a la imposibilidad de sincronizar al periférico con el periodo de la onda de AC debido a la incertidumbre de esta última (los 50/60Hz no son necesariamente estables). En contraste, el código de la PIO se traduce a un lenguaje tipo ensamblador que, una vez ensamblado, se inyecta al periférico para su ejecución autónoma.

Cada PIO cuenta con un PC o contador de programa, un registro de corrimiento de entrada o *ISR (input Shift Register)*, un registro de corrimiento de salida u *OSR (Output Shift Register)*, dos registros auxiliares `X` e `Y`, y una lógica de control; donde cada registro es de 32 bits. Los registros de corrimiento son accessibles mediante colas síncronas de entrada y salida de 4 niveles, no así los registros auxiliares.

[8] Toda declaración es una llamada implícita al constructor del objeto por lo que se hace uso de la pila con cada instrucción de Python. Además, cuando existe una relación de herencia, Python debe llamar al constructor de cada clase padre para instanciar al nuevo objeto, aumentando el número de *push* a la pila.

El autómata a cargar tiene siete estados como sigue:

q_0. **Inicialización y carga del retraso**
Se apaga el TRIAC (pin en cero) y se espera a que haya datos en el `OSR`.
Los datos del `OSR` (número de ciclos de espera) se copian al registro `X`.

rp2040-test-acdc.py:24-26

```
    set(pin, 0)
    pull()                    # Loads OSR with data
    mov(x, osr)               # puts OSR contents in X
```

q_1. **Bucle. Espera por el cruce en cero** Se espera por un flanco de bajada en el pin de cruce por cero, indicando que el voltaje ha caído y la senoidal está en transición.

rp2040-test-acdc.py:29-33

```
    label('waitzx')
    wait(0, pin, 0)
```

q_2. **Actualización del tiempo de espera**
Se actualiza el valor de retardo copiando los datos de la cola de transmisión al registro `OSR`, si los hubiere (si no, OSR queda como está).
Posteriormente los datos se transfieren a los registros auxiliares `X` e `Y`.

rp2040-test-acdc.py:36-38

```
    pull(noblock)             # Loads OSR with data
    mov(x, osr)               # puts OSR contents in X
    mov(y, x)                 # puts X contents in Y
```

q_3. **Espera de transición**
Se espera por un flanco de subida en el pin de cruce por cero, lo que indica que la transición ha terminado y el voltaje vuelve a subir. Posteriormente se realiza una espera de 4 ciclos de reloj (800μs) a fin de que haya suficiente voltaje para poder habilitar el TRIAC.

rp2040-test-acdc.py:41-42

```
    wait(1, pin, 0)
    nop()                [4]
```

q_4. **Espera activa (variación de potencia)**
Se decrementa el registro `Y` hasta que llegue a cero.

rp2040-test-acdc.py:48-49

```
    label('delay')
    jmp(y_dec, 'delay')
```

q_5. **Envío del pulso de encendido al TRIAC**
Se envía un pulso de al menos 2μs al TRIAC para encenderlo.
Como cada instrucción dura 200μs (cien veces más) una instrucción es más que suficiente.

rp2040-test-acdc.py:52-53

```
    set(pins, 1)
    set(pins, 0)
```

q_6. **Salto al estado `q1`**
Se cierra el bucle regresando a la espera de cruce por cero.

rp2040-test-acdc.py:56-56

```
    jmp('waitzx')
```

La máquina de estados se configura para operar a 5kHz (200μs por instrucción), lo que da aproximadamente 21 pasos por periodo, una variación de potencia aproximada del 5 % por cada paso asumiendo una progresión lineal.

4.2. Paso 2: Configuración de comunicaciones I²C

La comunicación está dividida en dos etapas: la configuración del dispositivo maestro (la Raspberry Pi) y la configuración del dispositivo esclavo (el RP2040). Se comenzará con la configuración del dispositivo maestro.

Configuración del dispositivo maestro

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 #.

Código ejemplo 1: /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 python-smbus
# adduser pi i2c
```

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: -- -- -- -- -- -- -- --
```

Configuración del dispositivo esclavo

A continuación, es necesario configurar el RP2040 para funcionar como dispositivo esclavo o *slave* en el bus I²C.

Debido a que los desarrolladores de MicroPython consideran que el RP2040 es un dispositivo demasiado poderoso para actuar esclavo, el firmware sólo considera la configuración del periférico como maestro. Así, será necesario configurar a nivel registro el periférico utilizando las primitivas de acceso directo a memoria `machine.mem32` de

MircoPython y realizar todas las transacciones a nivel registro y sin posibilidad de gestionar de forma eficiente las interrrupciones.

Por fortuna para los estudiantes, en este manual se incluye la librería `i2cslave.py` que realiza justamente eso. Para usarla, basta con cargar el archivo y crear un objeto de tipo `I2CSlave` y especificar el número de periférico a utilizar (I^2C0 o I^2C1) y los pines que se utilizarán como SDA y SCL, tal como se muestra en el Código de Ejemplo 2.

Código ejemplo 2: `rp2040-test-i2c.py:23` — Dirección asignada al dispositivo esclavo

```
1     i2c = I2CSlave(address=0x0A)
```

Al no existir soporte para interrupciones (las comunicaciones I^2C son demasiado rápidas para Python), será necesario realizar las operaciones de lectura y escritura mediante los métodos:

- `rxBufferCount()` que devuelve el número de bytes almacenados en el buffer de recepción,
- `read()` que realiza una lectura síncrona de los datos en el buffer, y
- `write(data)` que realiza una transmisión sincrona de los datos proporcionados como un tipo `bytes` o `bytearray`.

4.3. Paso 3: Control en lazo abierto de la potencia de una carga resistiva

Antes de proceder, verifique conexiones con un multímetro en busca de corto circuitos. En particular verifique que los circuitos de AC y DC funcionan de manera independiente y que existe una impedancia infinita entre pines optoacoplados.

Con la Raspberry Pi configurada, basta con generar los dos programas para transferir la potencia de salida deseada de la Raspberry Pi al RP2040 que se encargará cortar el flujo de corriente en el instante correcto para obtener la potencia deseada.

Primero, es necesario configurar al RP2040 como dispositivo esclavo e inicializar el bus I^2C, tal como se muestra en el Código de Ejemplo 3. Las comunicaciones via I^2C son síncronas, lo que simplifica enormemente el diseño del programa.

Código ejemplo 3: `rp2040-test-i2c.py:23` — Dirección asignada al dispositivo esclavo

```
1     i2c = I2CSlave(address=0x0A)
```

Tanto el envío como la recepción de datos se realizan byte por byte, por lo que es necesario convertir la potencia (*float*) en un arreglo de bytes que pueda ser transmitido. Para esta opreación se utilizará la librería `ustruct` que empaquetará y desempaquetará flotantes (*float*) en listas de 4 bytes que pueden ser enviados o recibidos via I^2C de manera análoga a como se muestra en los Códigos de Ejemplo 5 y 6

Del lado de la Raspberry Pi, primero ha inicializarse el bus I^2C 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 4).

Código ejemplo 4: `raspberry-code-i2c.py:18` — Configuración del bus I^2C

```
1 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` que empaquetará y desempaquetará flotantes (*float*) en listas de 4 bytes que pueden ser enviados o recibidos del RP2040 via I^2C tal como se muestra en los Códigos de Ejemplo 5 y 6

Código ejemplo 5: `raspberry-code-i2c.py:36-43` — Escritura de flotantes en el bus I^2C

```
1 def writePower(pwr):
2   try:
3     data = struct.pack('<f', pwr) # Packs number as float
4     # Creates a message object to write 4 bytes from SLAVE_ADDR
5     msg = smbus2.i2c_msg.write(SLAVE_ADDR, data)
6     i2c.i2c_rdwr(msg)  # Performs write
7   except:
8     pass
```

Código ejemplo 6: `raspberry-code-i2c.py:20-34` — Lectura de flotantes del bus I^2C

```
def readPower():
  try:
    # Creates a message object to read 4 bytes from SLAVE_ADDR
    msg = smbus2.i2c_msg.read(SLAVE_ADDR, 4)
    i2c.i2c_rdwr(msg)  # Performs write
    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))
    pwr = struct.unpack('<f', ba)
    # print('Received power: {} = {}'.format(data, pwr))
    return pwr
  except:
    return -1
```

El resto del programa es trivial, pues consiste sólo en solicitar al usuario un valor de potencia y enviarlo al RP2040.

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

5. Experimentos

1. [6pt] Alambre el circuito completo y combine el código de los Apéndices A a E para poder controlar la intensidad del brillo del foco incandecente con la Raspberry Pi usando los valores tecleados en la consola (porentaje de 0–100 % de la potencia total).

2. [4pt] Modifique el código anterior para que la consola presente la potencia real modulada por el RP2040 (equivalente en potencia del tiempo de encendido en milisegundos).

3. [+3pt] Modifique el código del punto 2 para que el arduino pueda modular la potencia del foco incandecente entre 0 % y 100 % con una resolución máxima de 1 %. Imprima la potencia reportada por el arduino con un dígito decimal. Justifique el código modigificado con los cálculos pertinentes que prueben que los valores reportados son correctos dentro de la resolución solicitada.

4. [+2pt] Con base en lo aprendido, modifique el código delpunto 3 para que la Raspberry Pi sirva una página web donde se pueda modificar con un control gráfico la potencia de encendido del foco.

6. Referencias

Referencias

[1] *4N25 Optocoupler, Phototransistor Output, with Base Connection.* Vishay Semiconductors, 8 2010. Revised: January, 2010.

[2] Muhammad H Rashid. Power electronic, devices, circuits, and applications. *Handbook, Second Edition, Burlington*, 2006.

[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] *MOC3021 Random-Phase Optoisolator TRIAC Driver Output.* Fairchild Semiconductors, 8 2010. Revised: January, 2010.

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

[6] 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: `rp2040-test-zx.py`

src/rp2040–test–zx.py

```
from machine import Pin, Timer
from utime import sleep_us

zxcount = 0
flag    = False
zxpin   = 0
timer   = None


def setup():
    global zxpin, timer
    zxpin  = Pin(zxpin, Pin.IN)
    zxpin.irq(trigger=Pin.IRQ_FALLING, handler=zxhandle)
    timer  = Timer(period=1000, mode=Timer.PERIODIC, callback=timerHandle)
#end def


def main():
    setup()
    print('Running')
    while True:
        mainloop()
#end def


def mainloop():
    global flag, zxcount
    if not flag:
        # Slack until next interrupt
        sleep_us(10)
        return
    flag = not flag
    print('Count: ', zxcount)
    zxcount = 0
#end def


def zxhandle(pin):
    global zxcount
    zxcount+= 1
#end def


def timerHandle(tmr):
    global flag
    flag = True
#end def

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

B. Programa Ejemplo: `rp2040-test-i2c.py`

src/rp2040–test–i2c.py

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

# Prints all floats arriving thorugh i2c (slave)
# and replies doubling that number
def main():
    i2c = I2CSlave(address=0x0A)
    print('Slave ready')

    while True:
        # Waits until there are exactly 4 bytes (1 float) in buffer
        while i2c.rxBufferCount() < 4:
            sleep_us(10)
        data = i2c.read()
        pwr = ustruct.unpack("<f", data)
        data = ustruct.pack('<f', pwr * 2)
        i2c.write(data)
        print(f'Master said: {pwr}')
# end def

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

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

src/raspberry–code–i2c.py

```python
import smbus2
import struct
import time

# Arduino's I2C device address
SLAVE_ADDR = 0x0A # I2C Address of RP2040

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

def readPower():
  try:
    # Creates a message object to read 4 bytes from SLAVE_ADDR
    msg = smbus2.i2c_msg.read(SLAVE_ADDR, 4)
    i2c.i2c_rdwr(msg)  # Performs write
    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))
    pwr = struct.unpack('<f', ba)
    # print('Received power: {} = {}'.format(data, pwr))
    return pwr
  except:
    return -1

def writePower(pwr):
  try:
    data = struct.pack('<f', pwr) # Packs number as float
    # Creates a message object to write 4 bytes from SLAVE_ADDR
    msg = smbus2.i2c_msg.write(SLAVE_ADDR, data)
    i2c.i2c_rdwr(msg)  # Performs write
  except:
    pass

def main():
  while True:
    try:
      power = input("Power? ")
      power = float(power)
      if power >= 0 and power <= 100:
        writePower(power)
        print("\tPower set to {}".format(readPower()))
      else:
        print("\tInvalid!")
    except:
      print("\tInvalid!")

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

D. Programa Ejemplo: `rp2040-test-acdc.py`

src/rp2040–test–zx.py

```
from machine import Pin, Timer
from utime import sleep_us

zxcount = 0
flag    = False
zxpin   = 0
timer   = None


def setup():
    global zxpin, timer
    zxpin  = Pin(zxpin, Pin.IN)
    zxpin.irq(trigger=Pin.IRQ_FALLING, handler=zxhandle)
    timer  = Timer(period=1000, mode=Timer.PERIODIC, callback=timerHandle)
#end def


def main():
    setup()
    print('Running')
    while True:
        mainloop()
#end def


def mainloop():
    global flag, zxcount
    if not flag:
        # Slack until next interrupt
        sleep_us(10)
        return
    flag = not flag
    print('Count: ', zxcount)
    zxcount = 0
#end def


def zxhandle(pin):
    global zxcount
    zxcount+= 1
#end def


def timerHandle(tmr):
    global flag
    flag = True
#end def

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

E. Librería: `i2cslave.py`

src/i2cslave.py

```
import machine
from utime import sleep_ms, sleep_us

# Constants from the RP2040 Datasheet
__IO_BANK0_BASE          = 0x40014000 # GPIO registers base addresses
__I2C0_BASE              = 0X40044000 # I2C0 registers base addresses
__I2C1_BASE              = 0X40048000 # I2C1 registers base addresses
__ATOM_RW                = 0x0000
__ATOM_XOR               = 0x1000
__ATOM_SET               = 0x2000
__ATOM_CLR               = 0x3000
__IC_CON                 = 0x00 # I2C Control Register
__IC_TAR                 = 0x04 # I2C Target Address Register
__IC_SAR                 = 0x08 # I2C Slave Address Register
__IC_DATA_CMD            = 0x10 # I2C Rx/Tx Data Buffer and Command Register
__IC_SS_SCL_HCNT         = 0x14 # Standard Speed I2C Clock SCL High Count Register
__IC_SS_SCL_LCNT         = 0x18 # Standard Speed I2C Clock SCL Low Count Register
__IC_FS_SCL_HCNT         = 0x1c # Fast Mode or Fast Mode Plus I2C Clock SCL High Count
    Register
__IC_FS_SCL_LCNT         = 0x20 # Fast Mode or Fast Mode Plus I2C Clock SCL Low Count
    Register
__IC_INTR_STAT           = 0x2c # I2C Interrupt Status Register
__IC_INTR_MASK           = 0x30 # I2C Interrupt Mask Register
__IC_RAW_INTR_STAT       = 0x34 # I2C Raw Interrupt Status Register
__IC_RX_TL               = 0x38 # I2C Receive FIFO Threshold Register
__IC_TX_TL               = 0x3c # I2C Transmit FIFO Threshold Register
__IC_CLR_INTR            = 0x40 # Clear Combined and Individual Interrupt Register
__IC_CLR_RX_UNDER        = 0x44 # Clear RX_UNDER Interrupt Register
__IC_CLR_RX_OVER         = 0x48 # Clear RX_OVER Interrupt Register
__IC_CLR_TX_OVER         = 0x4c # Clear TX_OVER Interrupt Register
__IC_CLR_RD_REQ          = 0x50 # Clear RD_REQ Interrupt Register
__IC_CLR_TX_ABRT         = 0x54 # Clear TX_ABRT Interrupt Register
__IC_CLR_RX_DONE         = 0x58 # Clear RX_DONE Interrupt Register
__IC_CLR_ACTIVITY        = 0x5c # Clear ACTIVITY Interrupt Register
__IC_CLR_STOP_DET        = 0x60 # Clear STOP_DET Interrupt Register
__IC_CLR_START_DET       = 0x64 # Clear START_DET Interrupt Register
__IC_CLR_GEN_CALL        = 0x68 # Clear GEN_CALL Interrupt Register
__IC_ENABLE              = 0x6c # I2C ENABLE Register
__IC_STATUS              = 0x70 # I2C STATUS Register
__IC_TXFLR               = 0x74 # I2C Transmit FIFO Level Register
__IC_RXFLR               = 0x78 # I2C Receive FIFO Level Register
__IC_SDA_HOLD            = 0x7c # I2C SDA Hold Time Length Register
__IC_TX_ABRT_SOURCE      = 0x80 # I2C Transmit Abort Source Register
__IC_SLV_DATA_NACK_ONLY  = 0x84 # Generate Slave Data NACK Register
__IC_DMA_CR              = 0x88 # DMA Control Register
__IC_DMA_TDLR            = 0x8c # DMA Transmit Data Level Register
__IC_DMA_RDLR            = 0x90 # DMA Transmit Data Level Register
__IC_SDA_SETUP           = 0x94 # I2C SDA Setup Register
__IC_ACK_GENERAL_CALL    = 0x98 # I2C ACK General Call Register
__IC_ENABLE_STATUS       = 0x9c # I2C Enable Status Register
__IC_FS_SPKLEN           = 0xa0 # I2C SS, FS or FM+ spike suppression limit
__IC_CLR_RESTART_DET     = 0xa8 # Clear RESTART_DET Interrupt Register
__IC_COMP_PARAM_1        = 0xf4 # Component Parameter Register 1
__IC_COMP_VERSION        = 0xf8 # I2C Component Version Register
__IC_COMP_TYPE           = 0xfc # I2C Component Type Register

class I2CSlave():
#'''
# This class allows to use the RP2040 as an I2C slave device.
# Since there is no support for this in MicroPython, we do this
# at register-level with machine.memXX[ADDR] = Value
#'''


```

```
63  def __init__(self, id=0, address=0x27, sda=None, scl=None):
64    if id > 1: raise ValueError('Unsupported')
65    elif id == 0:
66      if sda is None: sda = 0 #sda = machine.Pin(0)
67      if scl is None: scl = 1 #scl = machine.Pin(1)
68      if sda not in [0, 4, 8, 12, 16, 20]:
69        raise ValueError('Invalid pin number for sda')
70      if scl not in [1, 5, 9, 13, 17, 21]:
71        raise ValueError('Invalid pin number for scl')
72    else: # id == 1:
73      if sda is None: sda = 2 #sda = machine.Pin(2)
74      if scl is None: scl = 3 #scl = machine.Pin(3)
75      if sda not in [2, 6, 10, 14, 18, 26]:
76        raise ValueError('Invalid pin number for sda')
77      if scl not in [3, 7, 11, 15, 19, 27]:
78        raise ValueError('Invalid pin number for scl')
79
80    if address < 0 or address > 0x7f:
81      raise ValueError('Address out of range (7bit addresses only)')
82
83    self._sda  = sda
84    self._scl  = scl
85    self._addr = address
86    self._base = __I2C0_BASE if id == 0 else __I2C1_BASE
87
88    # Setup pins
89    self.__setupPin(sda)
90    self.__setupPin(scl)
91
92    # From pp 453
93    # 1. Disable the DW_apb_i2c by writing a '0' to IC_ENABLE.ENABLE (bit 0).
94    self.__regClr(__IC_ENABLE, 0x0001)
95    # 2. Write to the IC_SAR register (bits 9:0) to set the slave address.
96    self.__regClr(__IC_SAR, 0x01ff)
97    self.__regSet(__IC_SAR, address)
98
99    # 3. Set Configuration (pp 466, bits 7-0)
100   # Clear bits 6, 3 and 0 (Disable slave, 10 bits, Master mode)
101   self.__regClr(__IC_CON, 0x0049)
102
103
104   # 4. Re-enable I2C  (bit 0)
105   self.__regSet(__IC_ENABLE, 0x0001)
106
107   self.__initialized = True
108 # end def
109
110 def __setupPin(self, pin):
111   # Page 243, Table 282: Each GPIO pin uses 8 bytes: 4 status, 4 control
112   # To setup we clear and set the 32-bit Control register. Offset = 4
113   gpioaddr = __IO_BANK0_BASE + 8 * pin + 4
114   machine.mem32[ gpioaddr | __ATOM_CLR ] = 0x1f # FUNCSEL mask
115   machine.mem32[ gpioaddr | __ATOM_SET ] = 0x03 # Attach pin to I2C
116 # end def
117
118 def __regClr(self, reg, mask):
119   machine.mem32[self._base | __ATOM_CLR | reg] = mask
120 # end def
121
122 def __regSet(self, reg, mask):
123   machine.mem32[self._base | __ATOM_SET | reg] = mask
124 # end def
125
126 def __regRead(self, reg, andmask=0xffffffff):
127   return machine.mem32[self._base | __ATOM_RW | reg] & andmask
128
129 def __regWrite(self, reg, value):
130   machine.mem32[self._base | __ATOM_RW  | reg] = value
```

```
131   # end def
132
133   def __regXor(self, reg, mask):
134     machine.mem32[self._base | __ATOM_XOR | reg] = mask
135   # end def
136
137   @property
138   def id(self):
139     return 0 if self._base == __I2C0_BASE else 1
140   # end def
141
142   @property
143   def sda(self):
144     return self._sda
145   # end def
146
147   @property
148   def scl(self):
149     return self._scl
150   # end def
151
152   @property
153   def address(self):
154     return self._addr
155   # end def
156
157
158   def deInit(self):
159     self.__initialized = False
160     #'''Turn off the I2C bus.'''
161     # 1. Disable the DW_apb_i2c by writing a '0' to IC_ENABLE.ENABLE (bit 0).
162     self.__regClr(__IC_ENABLE, 0x0001)
163     # 2. Clear the IC_SAR register
164     self.__regClr(__IC_SAR, 0x01ff)
165     # 3. Clear Configuration (pp 466, bits 7-0)
166     self.__regClr(__IC_CON, 0x0049)
167     # Set bits 6 and 0 (Disable slave, Master mode)
168     self.__regSet(__IC_CON, 0x0021)
169     # 4. Re-enable I^2C  (bit 0)
170     self.__regSet(__IC_ENABLE, 0x0001)
171     self._sda  = None
172     self._scl  = None
173     self._addr = None
174     self._base = None
175   # end def
176
177   def idle(self):
178     return not self.__regRead(__IC_STATUS, 0x01)
179   # end def
180
181   def rxBufferCount(self):
182     return self.__regRead(__IC_RXFLR, 0x1f)
183   # end def
184
185   def rxBufferEmpty(self):
186     return not self.__regRead(__IC_STATUS, 0x08)
187   # end def
188
189   def rxBufferFull(self):
190     return self.__regRead(__IC_STATUS, 0x10)
191   # end def
192
193   def txBufferCount(self):
194     return self.__regRead(__IC_TXFLR, 0x10)
195   # end def
196
197   def txBufferEmpty(self):
198     return not self.__regRead(__IC_STATUS, 0x04)
```

```
199   # end def
200
201   def txBufferFull(self):
202     return not self.__regRead(__IC_STATUS, 0x02)
203   # end def
204
205   def read(self):
206     #'''Blocks until the Master sends some data,
207     #   then retrieves the data from the buffer'''
208     if not self.__initialized: raise IOError('Uninitialized')
209     # Wait until data arrives to the buffer
210     while not self.__regRead(__IC_STATUS, 0x08) and self.__regRead(__IC_RXFLR, 0x1f) < 1:
211       sleep_us(10)
212     # Get number of bytes in buffer, allocate memory and retrieve
213     bytecount = self.__regRead(__IC_RXFLR, 0x1f)
214     ba = bytearray(bytecount)
215     for i in range(bytecount):
216       ba[i] = self.__regRead(__IC_DATA_CMD, 0xff)
217     return ba
218   # end def
219
220   def readByte(self):
221     #'''Blocks until the Master sends some data,
222     #   then retrieves the data from the buffer'''
223     if not self.__initialized: raise IOError('Uninitialized')
224     # Wait until data arrives to the buffer
225     while not self.__regRead(__IC_STATUS, 0x08):
226       sleep_us(10)
227     # Return first byte in the buffer
228     return self.__regRead(__IC_DATA_CMD, 0xff)
229   # end def
230
231   def write(self, ba):
232     #'''Blocks until the Master requests data, then writes the data
233     #   in the buffer. Lags until all the data has been written.
234     #   We can't just put the data in the txBuffer and return (async)
235     #   bc upon the RD_REQ arrival a TX_ABRT is generated and the
236     #   txBuffer is automatically flushed (see 4.3.10.1.2, pp. 454)'''
237     if not self.__initialized: raise IOError('Uninitialized')
238     if not isinstance(ba, (bytes, bytearray)): raise ValueError('An array of bytes is
      required')
239     for b in ba:
240       self.writeByte(b)
241   # end def
242
243   def writeByte(self, b):
244     #'''Blocks until the Master requests byte, then writes the byte
245     #   in the buffer.
246     #   We can't just put the data in the txBuffer and return (async)
247     #   bc upon the RD_REQ arrival a TX_ABRT is generated and the
248     #   txBuffer is automatically flushed (see 4.3.10.1.2, pp. 454)'''
249     if not self.__initialized: raise IOError('Uninitialized')
250     # 1. Wait for the RD_REQ signal (bit 5)
251     while not self.__regRead(__IC_RAW_INTR_STAT, 0x20):
252       sleep_us(10)
253     # 2. Clear the ABORT register that would abort a transmission
254     self.__regClr(__IC_CLR_TX_ABRT, 0x01)
255     # 3. Clear the read request by reading this register
256     self.__regRead(__IC_CLR_RD_REQ)
257     # 4. Wait until there is space in the Tx Buffer
258     while not self.__regRead(__IC_STATUS, 0x02):
259       sleep_us(10)
260     self.__regWrite(__IC_DATA_CMD, b & 0xff)
261   # end def
262
263   def waitForData(self, timeout=-1):
264     # '''Blocks until data is received in the rxBuffer
265     #    Returns true if data arrived before the timeout, false otherwise.
```

```
266     # '''
267     if not self.__initialized: raise IOError('Uninitialized')
268     while self.rxBufferEmpty() and (timeout > 0):
269       timeout-= 1
270       sleep_ms(1)
271     return not self.rxBufferEmpty()
272   # end def
273
274   def waitForRdReq(self, timeout=-1):
275     # '''Awaits for a read request for up to timeout ms.
276     #    Returns true if the read requests arrived, false otherwise.
277     # '''
278     if not self.__initialized: raise IOError('Uninitialized')
279     while (self.__regRead(__IC_RAW_INTR_STAT, 0x20) != 0) and (timeout > 0):
280       timeout-= 1
281       sleep_ms(1)
282     return self.__regRead(__IC_RAW_INTR_STAT, 0x20)
283   # end def
284 # end class
```