

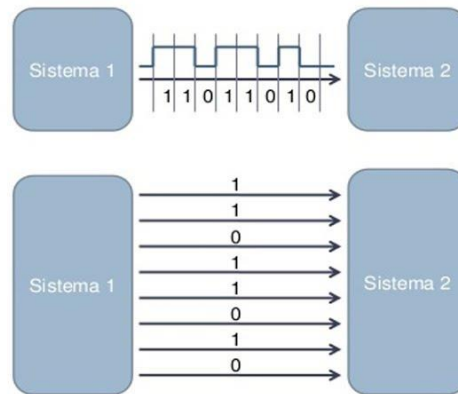
Comunicaciones

Contenido

1. Comunicación serie	1
1.1. Introduccion	12
1.2. Reglas de serial	13
1.3. cableado y Hardware	12
2. Interfaz de periféricos en serie (SPI)	Error! Bookmark not defined.
2.1. Introduccion	12
2.2. ¿Qué hay de malo con los puertos serie?	13
2.3. Una solución sincrónica	12
2.1. Recibiendo información.....	12
2.2. Selección de esclavo (SS)	13
2.3. Varios esclavos	12
3.Programación para SPI	12
3.1. SPI y entrada analógica.....	12
3.2. Conexiones de hardware.....	13
3.3. Código: Lectura de voltaje analógico	21
3.4. Desafío	22
4. I2C	19
4.1 Introducción	19
4.2¿Por qué utilizar I2C?	20
4.3¿Qué hay de malo con los puertos serie UART?	20
4.4 ¿Qué está mal con SPI?	21
4.5 I ² C - ¡Lo mejor de ambos mundos!	22
4.6 I ² C - Una breve historia.....	22
4.7 Protocolo	23
4.8 Temas de protocolo avanzado	24
5. Sensor de temperatura I2C	26
5.1. Conexiones de hardware	12
5.2. Reglas de serial	13
5.3. Código: leer y calcular la temperatura	12
5.2. Desafío:	13
6.conectar Arduino y Raspberry Pi por comunicación I2C	30

1.Comunicación serie

Comunicación serial

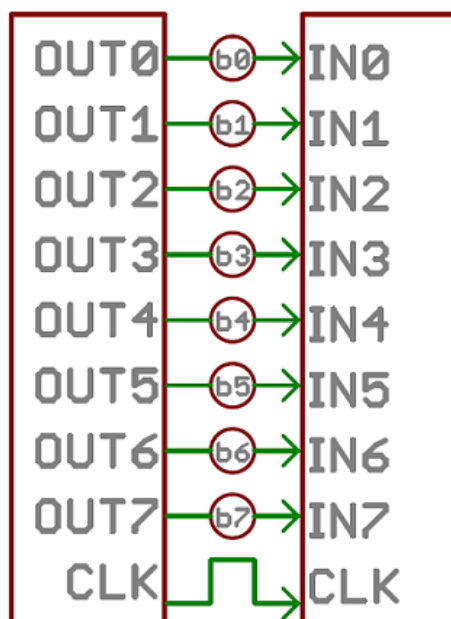


1.1. Introduccion

La electrónica integrada se trata de interconectar circuitos (procesadores u otros circuitos integrados) para crear un sistema simbiótico. Para que esos circuitos individuales intercambien su información, deben compartir un protocolo de comunicación común. Se han definido cientos de protocolos de comunicación para lograr este intercambio de datos y, en general, cada uno se puede separar en una de dos categorías: paralelo o serie.

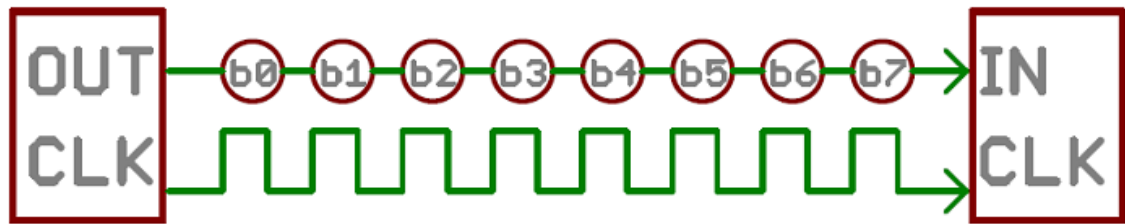
Paralelo frente a serie

Las interfaces paralelas transfieren varios bits al mismo tiempo. Por lo general, requieren **buses** de datos, que se transmiten a través de ocho, dieciséis o más cables. Los datos se transfieren en olas enormes y estrepitosas de 1 y 0.



Un bus de datos de 8 bits, controlado por un reloj, que transmite un byte por cada pulso de reloj. Se utilizan 9 cables.

Las interfaces seriales transmiten sus datos, un bit a la vez. Estas interfaces pueden operar con tan solo un cable, generalmente nunca más de cuatro.



Ejemplo de una interfaz en serie, que transmite un bit por cada pulso de reloj. ¡Solo se requieren 2 cables!

Piense en las dos interfaces como un flujo de automóviles: una interfaz paralela sería la megaautopista de más de 8 carriles, mientras que una interfaz en serie se parece más a una ruta rural de dos carriles. Durante un período de tiempo determinado, la megaautopista potencialmente lleva a más personas a sus destinos, pero esa vía rural de dos carriles cumple su propósito y cuesta una fracción de los fondos para construir. La comunicación paralela ciertamente tiene sus beneficios. Es rápido, sencillo y relativamente fácil de implementar. Pero requiere muchas más líneas de entrada / salida (E / S).

Serie asincrónica

A lo largo de los años, se han diseñado decenas de protocolos en serie para satisfacer las necesidades particulares de los sistemas integrados. USB (bus *serie* universal) y Ethernet son algunas de las interfaces seriales informáticas más conocidas. Otras interfaces seriales muy comunes incluyen SPI, I²C y el estándar serial del que estamos aquí para hablar hoy. Cada una de estas interfaces seriales se puede clasificar en uno de dos grupos: síncrona o asíncrona.

Una interfaz serial síncrona siempre empareja su (s) línea (s) de datos con una señal de reloj, por lo que todos los dispositivos en un bus serial síncrono comparten un reloj común. Esto hace que la transferencia en serie sea más sencilla y, a menudo, más rápida, pero también requiere al menos un cable adicional entre los dispositivos de comunicación. Ejemplos de interfaces síncronas incluyen SPI e I²C.

Asíncrono significa que los datos se transfieren **sin el apoyo de una señal de reloj externa** . Este método de transmisión es perfecto para minimizar los cables y los pines de E / S necesarios, pero significa que debemos hacer un esfuerzo adicional para transferir y recibir datos de manera confiable. El protocolo serial que discutiremos en este tutorial es la forma más común de transferencias asincrónicas. De hecho, es tan común que cuando la mayoría de la gente dice "serial" están hablando de este protocolo (algo que probablemente notará a lo largo de este tutorial).

1.2. Reglas de serial

El protocolo serial asincrónico tiene una serie de reglas integradas, mecanismos que ayudan a garantizar transferencias de datos sólidas y sin errores. Estos mecanismos, que obtenemos para evitar la señal de reloj externo, son:

- Bits de datos,
- Bits de sincronización,
- Bits de paridad,
- y tasa de baudios.

A través de la variedad de estos mecanismos de señalización, encontrará que no hay una sola forma de enviar datos en serie. El protocolo es altamente configurable. La parte fundamental es asegurarse de que **ambos dispositivos en un bus serie estén configurados para usar exactamente los mismos protocolos**.

Tasa de baudios

La velocidad en baudios especifica **qué tan rápido** se envían los datos a través de una línea serial. Por lo general, se expresa en unidades de bits por segundo (bps). Si invierte la velocidad en baudios, puede averiguar cuánto tiempo se tarda en transmitir un solo bit. Este valor determina cuánto tiempo el transmisor mantiene una línea serial alta / baja o en qué período el dispositivo receptor muestrea su línea.

Las velocidades en baudios pueden tener casi cualquier valor dentro de lo razonable. El único requisito es que ambos dispositivos funcionen a la misma velocidad. Una de las velocidades en baudios más comunes, especialmente para cosas simples donde la velocidad no es crítica, es **9600 bps**. Otros baudios "estándar" son 1200, 2400, 4800, 19200, 38400, 57600 y 115200.

Cuanto mayor es la velocidad en baudios, más rápido se envían / reciben los datos, pero existen límites en cuanto a la rapidez con la que se pueden transferir los datos. Por lo general, no verá velocidades superiores a 115200, eso es rápido para la mayoría de los microcontroladores. Si sube demasiado, comenzará a ver errores en el extremo receptor, ya que los relojes y los períodos de muestreo simplemente no pueden mantenerse al día.

Enmarcando los datos

Cada bloque (generalmente un byte) de datos transmitidos se envía en realidad en un *paquete* o *trama* de bits. Los marcos se crean agregando bits de sincronización y paridad a nuestros datos.



Un marco en serie. Algunos símbolos de la trama tienen tamaños de bits configurables.

Entremos en los detalles de cada una de estas piezas del marco.

Fragmento de datos

La esencia real de cada paquete en serie son los datos que transporta. De manera ambigua, llamamos *fragmento a* este bloque de datos, porque su tamaño no se indica específicamente. La cantidad de datos en cada paquete se puede establecer entre 5 y 9 bits. Ciertamente, el tamaño de datos estándar es su byte básico de 8 bits, pero otros tamaños tienen sus usos. Un fragmento de datos de 7 bits puede ser más eficiente que el de 8, especialmente si solo está transfiriendo caracteres ASCII de 7 bits.

Después de acordar la longitud de un carácter, ambos dispositivos seriales también deben acordar el carácter **final** de sus datos. ¿Se envían los datos del bit más significativo (msb) al mínimo, o viceversa? Si no se indica lo contrario, normalmente puede asumir que los datos se transfieren **primero con el bit menos significativo (lsb)**.

Bits de sincronización

Los bits de sincronización son dos o tres bits especiales transferidos con cada fragmento de datos. Son el **bit de inicio** y el **bit de parada**. Fieles a su nombre, estos bits marcan el comienzo y el final de un paquete. Siempre hay un solo bit de inicio, pero el número de bits de parada se puede configurar en uno o dos (aunque normalmente se deja en uno).

El bit de inicio siempre se indica mediante una línea de datos inactiva que va de 1 a 0, mientras que los bits de parada volverán al estado inactivo manteniendo la línea en 1.

Bits de paridad

La paridad es una forma de verificación de errores muy simple y de bajo nivel. Viene en dos sabores: par o impar. Para producir el bit de paridad, se suman todos los 5-9 bits del byte de datos y la uniformidad de la suma decide si el bit está establecido o no. Por ejemplo, asumiendo que la paridad se establece en par y se agrega a un byte de datos como 0b01011101, que tiene un número impar de 1's (5), el bit de paridad se establecería en 1. Por el contrario, si el modo de paridad se estableció en impar, el bit de paridad sería 0.

La paridad es *opcional* y no se usa mucho. Puede ser útil para transmitir a través de medios ruidosos, pero también ralentizará un poco la transferencia de datos y requiere que tanto el remitente como el receptor implementen el manejo de errores (por lo general, los datos recibidos que fallan deben volver a enviarse).

9600 8N1 (un ejemplo)

9600 8N1 - 9600 baudios, 8 bits de datos, sin paridad y 1 bit de parada - es uno de los protocolos seriales más utilizados. Entonces, ¿cómo serían uno o dos paquetes de datos 9600 8N1? ¡Pongamos un ejemplo!

Un dispositivo que transmita los caracteres [ASCII](#) 'O' y 'K' tendría que crear dos paquetes de datos. El valor ASCII de O (que es en mayúsculas) es 79, que se descompone en un valor binario de 8 bits de 01001111, mientras que el valor binario de K es 01001011. Todo lo que queda es agregar bits de sincronización.

No se indica específicamente, pero se supone que los datos se transfieren primero con el bit menos significativo. Observe cómo se envía cada uno de los dos bytes a medida que se lee de derecha a izquierda.



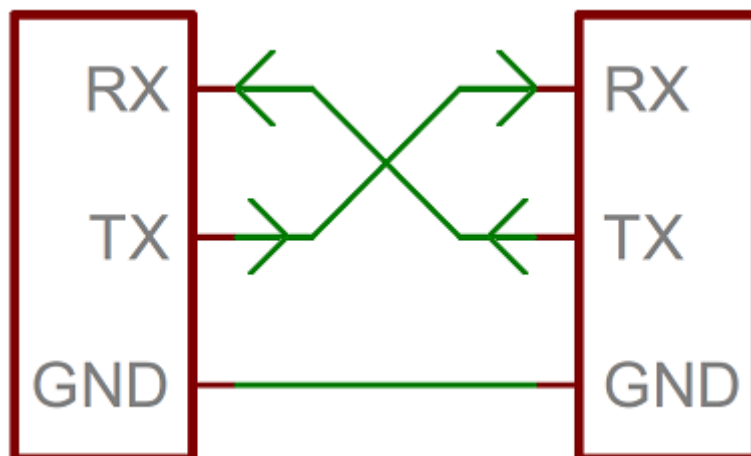
Dado que estamos transfiriendo a 9600 bps, el tiempo empleado en mantener cada uno de esos bits alto o bajo es $1 / (9600 \text{ bps})$ o $104 \mu\text{s}$ por bit.

Por cada byte de datos transmitidos, en realidad se envían 10 bits: un bit de inicio, 8 bits de datos y un bit de parada. Entonces, a 9600 bps, en realidad estamos enviando 9600 bits por segundo o 960 (9600/10) bytes por segundo.

Ahora que sabe cómo construir paquetes seriales, podemos pasar a la sección de hardware. ¡Allí veremos cómo esos unos y ceros y la velocidad en baudios se implementan a nivel de señal!

1.3. Cableado y hardware

Un bus serie consta de solo dos cables: uno para enviar datos y otro para recibir. Como tal, los dispositivos seriales deben tener dos pines seriales: el receptor, **RX**, y el transmisor, **TX**.



Es importante tener en cuenta que esas etiquetas **RX** y **TX** son con respecto al dispositivo en sí. Entonces, el RX de un dispositivo debe ir al TX del otro, y viceversa. Es extraño si estás acostumbrado a conectar VCC a VCC, GND a GND, MOSI a MOSI, etc., pero tiene sentido si lo piensas bien. El transmisor debe estar hablando con el receptor, no con otro transmisor.

Una interfaz en serie donde ambos dispositivos pueden enviar y recibir datos es **full-duplex** o **half-duplex**. Full-duplex significa que ambos dispositivos pueden enviar y recibir simultáneamente. La comunicación semidúplex significa que los dispositivos seriales deben turnarse para enviar y recibir.

Algunos buses seriales pueden salirse con la suya con una sola conexión entre un dispositivo de envío y de recepción. Por ejemplo, nuestras [pantallas LCD habilitadas en serie](#) son todo oídos y realmente no tienen ningún dato para transmitir al dispositivo de control. Esto es lo que se conoce como comunicación serial **simplex**. Todo lo que necesita es un solo cable desde el TX del dispositivo maestro hasta la línea RX del oyente.

2. Interfaz de periféricos en serie(SPI)

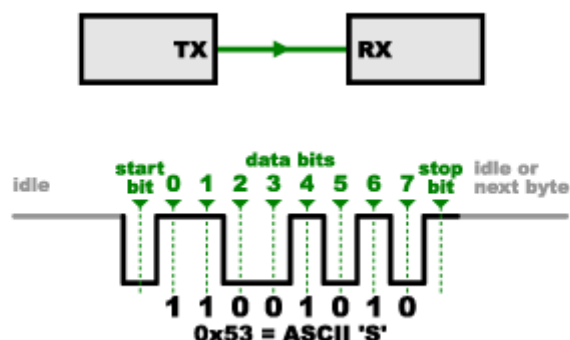
2.1 Introduccion

La interfaz de periféricos en serie (SPI) es un bus de interfaz que se usa comúnmente para enviar datos entre microcontroladores y pequeños periféricos, como registros de desplazamiento, sensores y tarjetas SD. Utiliza líneas de datos y reloj separadas, junto con una línea de selección para elegir el dispositivo con el que desea hablar.

2.2 ¿Qué hay de malo con los puertos serie?

Un puerto serie común, del tipo con líneas TX y RX, se denomina "asíncrono" (no síncrono) porque no hay control sobre cuándo se envían los datos ni garantía de que ambos lados estén funcionando exactamente a la misma velocidad. Dado que las computadoras normalmente dependen de que todo esté sincronizado con un solo "reloj" (el cristal principal conectado a una computadora que lo maneja todo), esto puede ser un problema cuando dos sistemas con relojes ligeramente diferentes intentan comunicarse entre sí.

Para solucionar este problema, las conexiones en serie asincrónicas agregan bits de inicio y parada adicionales a cada byte para ayudar al receptor a sincronizar los datos a medida que llegan. Ambas partes también deben acordar la velocidad de transmisión (como 9600 bits por segundo) de antemano. Las ligeras diferencias en la velocidad de transmisión no son un problema porque el receptor se vuelve a sincronizar al comienzo de cada byte.

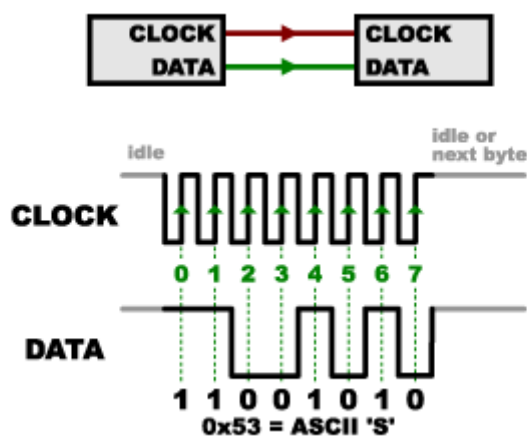


(Por cierto, si notó que "11001010" no es igual a 0x53 en el diagrama anterior, felicitaciones por su atención a los detalles. Los protocolos seriales a menudo envían los bits menos significativos primero, por lo que el bit más pequeño está en el extremo izquierdo. nybble inferior es en realidad 0011 = 0x3, y nybble superior es 0101 = 0x5.)

La serie asincrónica funciona bien, pero tiene mucha sobrecarga tanto en los bits de inicio y parada adicionales enviados con cada byte como en el hardware complejo requerido para enviar y recibir datos. Y como probablemente haya notado en sus propios proyectos, si ambos lados no están configurados a la misma velocidad, los datos recibidos serán basura. Esto se debe a que el receptor está muestreando los bits en momentos muy específicos (las flechas en el diagrama anterior). Si el receptor está mirando en los momentos incorrectos, verá los bits incorrectos.

2.3 Una solución sincrónica

SPI funciona de una manera ligeramente diferente. Es un bus de datos "síncrono", lo que significa que utiliza líneas separadas para los datos y un "reloj" que mantiene ambos lados en perfecta sincronización. El reloj es una señal oscilante que le dice al receptor exactamente cuándo muestrear los bits en la línea de datos. Este podría ser el borde ascendente (de menor a mayor) o descendente (de mayor a menor) de la señal del reloj; la hoja de datos especificará cuál usar. Cuando el receptor detecta ese borde, verá inmediatamente la línea de datos para leer el siguiente bit (vea las flechas en el diagrama de abajo). Debido a que el reloj se envía junto con los datos, especificar la velocidad no es importante, aunque los dispositivos tendrán una velocidad máxima a la que pueden operar (analizaremos la elección del borde y la velocidad del reloj adecuados en un momento).



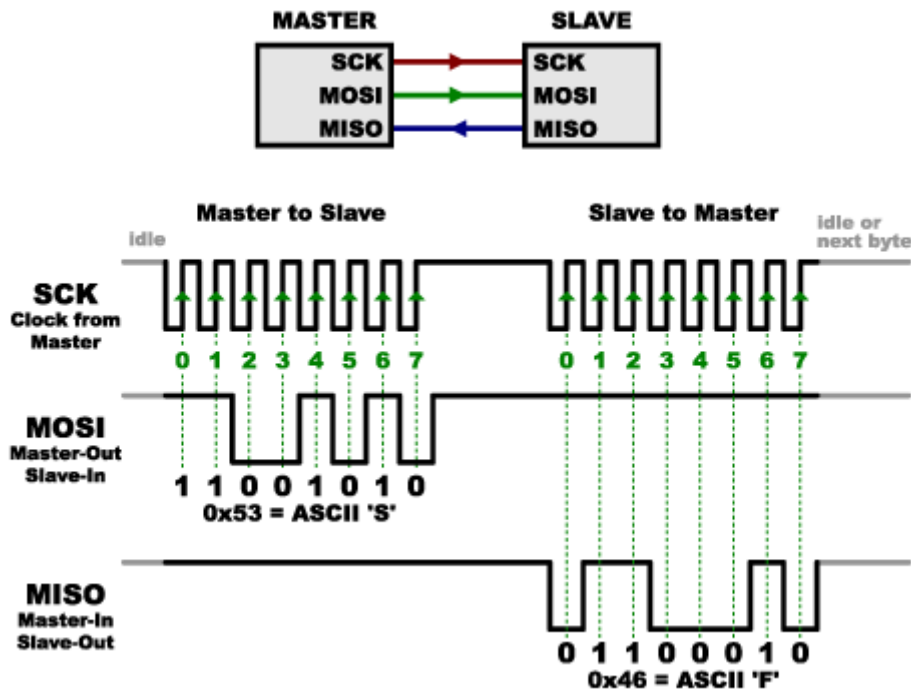
Una razón por la que SPI es tan popular es que el hardware receptor puede ser un simple [registro de desplazamiento](#). Esta es una pieza de hardware mucho más simple (¡y más barata!) que el UART (Receptor / Transmisor Asíncrono Universal) completo que requiere la serie asíncrona.

2.4 Recibiendo información

Es posible que esté pensando para sí mismo, que suena muy bien para las comunicaciones unidireccionales, pero ¿cómo envía datos en la dirección opuesta? Aquí es donde las cosas se complican un poco más.

En SPI, solo un lado genera la señal de reloj (generalmente llamado CLK o SCK para Serial Clock). El lado que genera el reloj se llama "maestro" y el otro lado se llama "esclavo". Siempre hay un solo maestro (que casi siempre es su microcontrolador), pero puede haber múltiples esclavos (más sobre esto en un momento).

Cuando los datos se envían desde el maestro a un esclavo, se envían en una línea de datos llamada MOSI, para "Master Out / Slave In". Si el esclavo necesita enviar una respuesta al maestro, el maestro continuará generando un número preestablecido de ciclos de reloj, y el esclavo colocará los datos en una tercera línea de datos llamada MISO, para "Master In / Slave Out".

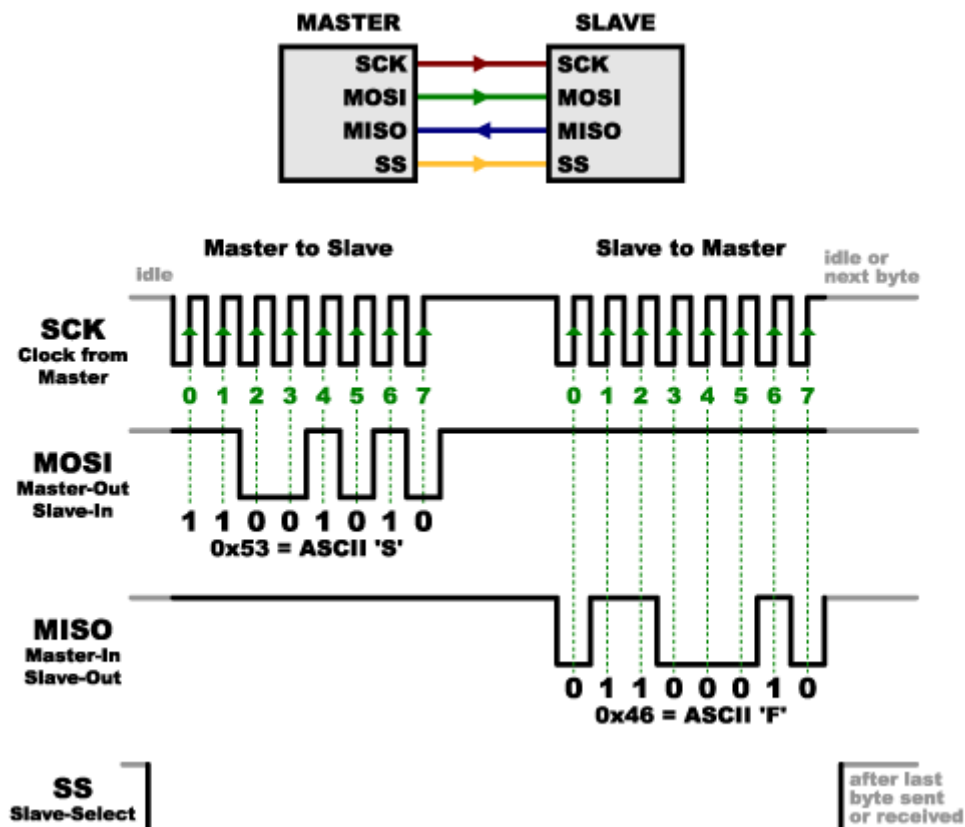


Observe que dijimos "arreglado previamente" en la descripción anterior. Debido a que el maestro siempre genera la señal de reloj, debe saber de antemano cuándo un esclavo necesita devolver datos y cuántos datos se devolverán. Esto es muy diferente a la serie asincrónica, donde se pueden enviar cantidades aleatorias de datos en cualquier dirección en cualquier momento. En la práctica, esto no es un problema, ya que SPI se usa generalmente para hablar con sensores que tienen una estructura de comando muy específica. Por ejemplo, si envía el comando para "leer datos" a un dispositivo, sabe que el dispositivo siempre le enviará, por ejemplo, dos bytes a cambio. (En los casos en los que desee devolver una cantidad variable de datos, siempre puede devolver uno o dos bytes especificando la longitud de los datos y luego hacer que el maestro recupere la cantidad completa).

Tenga en cuenta que SPI es "full duplex" (tiene líneas de envío y recepción separadas) y, por lo tanto, en ciertas situaciones, puede transmitir y recibir datos *al mismo tiempo* (por ejemplo, solicitar una nueva lectura del sensor mientras recupera los datos del anterior). La hoja de datos de su dispositivo le dirá si esto es posible.

2.5 Selección de esclavo (SS)

Hay una última línea que debe conocer, llamada SS para Selección de esclavos. Esto le dice al esclavo que debe despertarse y recibir / enviar datos y también se usa cuando hay varios esclavos presentes para seleccionar al que le gustaría hablar.

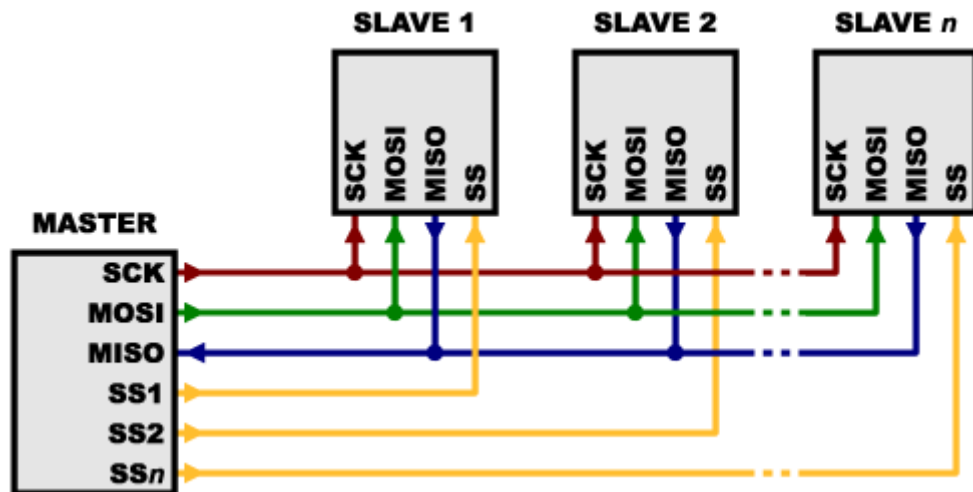


La línea SS normalmente se mantiene alta, lo que desconecta al esclavo del bus SPI. (Este tipo de lógica se conoce como "activo bajo", y lo verá a menudo utilizado para habilitar y restablecer líneas.) Justo antes de que los datos se envíen al esclavo, la línea se baja, lo que activa al esclavo. Cuando termine de usar el esclavo, la línea se vuelve alta nuevamente. En un [registro de desplazamiento](#), esto corresponde a la entrada "latch", que transfiere los datos recibidos a las líneas de salida.

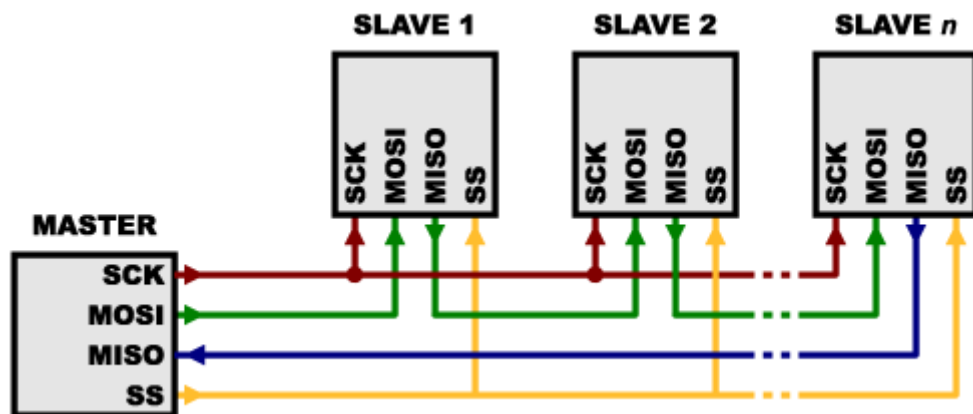
2.6 Varios esclavos

Hay dos formas de conectar varios esclavos a un bus SPI:

1. En general, cada esclavo necesitará una línea SS separada. Para hablar con un esclavo en particular, hará que la línea SS de ese esclavo sea baja y mantendrá el resto alto (no quiere que dos esclavos se activen al mismo tiempo, o ambos pueden intentar hablar en la misma línea MISO resultando en datos confusos). Muchos esclavos requerirán muchas líneas SS; Si se está quedando sin salidas, hay [chips decodificadores binarios](#) que pueden multiplicar sus salidas SS.



2. Por otro lado, algunas partes prefieren estar conectadas en cadena, con el MISO (salida) de uno yendo al MOSI (entrada) del siguiente. En este caso, una sola línea SS va a *todos* los esclavos. Una vez enviados todos los datos, se eleva la línea SS, lo que provoca que todos los chips se activen simultáneamente. Esto se usa a menudo para registros de cambios en cadena y [controladores LED direccionables](#).



Tenga en cuenta que, para este diseño, los datos se desborda de un esclavo de otro, por lo que enviar datos a cualquier *uno* de esclavos, que necesitará para transmitir datos suficientes para llegar a *todos* ellos. Además, tenga en cuenta que el *primer* dato que transmita terminará en el *último* esclavo.

Este tipo de diseño se usa generalmente en situaciones de solo salida, como LED de conducción donde no necesita recibir ningún dato. En estos casos, puede dejar desconectada la línea MISO del maestro. Sin embargo, si es necesario devolver los datos al maestro, puede hacerlo cerrando el bucle de conexión en cadena (cable azul en el diagrama anterior). Tenga en cuenta que si hace esto, los datos de retorno del esclavo 1 deberán pasar por *todos* los esclavos antes de volver al maestro, así que asegúrese de enviar suficientes comandos de recepción para obtener los datos que necesita.

3. Programación para SPI

Muchos microcontroladores tienen periféricos SPI integrados que manejan todos los detalles del envío y la recepción de datos y pueden hacerlo a velocidades muy altas. El protocolo SPI también es lo suficientemente simple como para que usted (¡sí, usted!) pueda escribir sus propias rutinas para manipular las líneas de E / S en la secuencia adecuada para transferir datos. (Un buen ejemplo está en la [página SPI de Wikipedia](#)).

Deberá seleccionar algunas opciones al configurar su interfaz. Estas opciones deben coincidir con las del dispositivo con el que está hablando; consulte la hoja de datos del dispositivo para ver qué requiere.

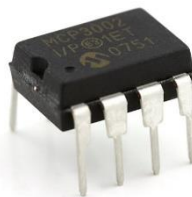
- La interfaz puede enviar datos con el bit más significativo (MSB) primero o el bit menos significativo (LSB) primero.
- El esclavo leerá los datos en el flanco ascendente o descendente del pulso de reloj. Además, el reloj se puede considerar "inactivo" cuando está alto o bajo.
- SPI puede funcionar a velocidades extremadamente altas (millones de bytes por segundo), lo que puede ser demasiado rápido para algunos dispositivos. Para adaptarse a estos dispositivos, puede ajustar la velocidad de datos.
- Si está utilizando la biblioteca SPI, debe usar los pines SCK, MOSI y MISO proporcionados, ya que el hardware está cableado a esos pines. También hay un pin SS dedicado que puede usar (que debe, al menos, estar configurado en una salida para que funcione el hardware SPI), pero tenga en cuenta que puede usar cualquier otro pin de salida disponible para SS su (s) dispositivo (s) esclavo también.

3.1. SPI y entrada analógica

Muchos sensores utilizan un voltaje analógico para transmitir sus datos de medición. Por ejemplo, las [fotocélulas](#) cambian su resistencia dependiendo de la cantidad de luz que caiga sobre el sensor. Al usar un divisor de voltaje en nuestro circuito, podemos medir de manera efectiva la cantidad de luz ambiental midiendo un voltaje.

La mala noticia es que nuestra Raspberry Pi no viene con ninguna forma de medir un voltaje analógico. Para hacer eso, necesitaremos confiar en una pieza separada de circuitos: un *convertidor de analógico a digital* (ADC). Específicamente, [usaremos](#) el [Microchip MCP3002](#) , que es un chip ingeniosamente pequeño que puede medir hasta 2 voltajes analógicos en canales separados e informar sus valores a través de la *interfaz Serial Peripheral Interface* (SPI).

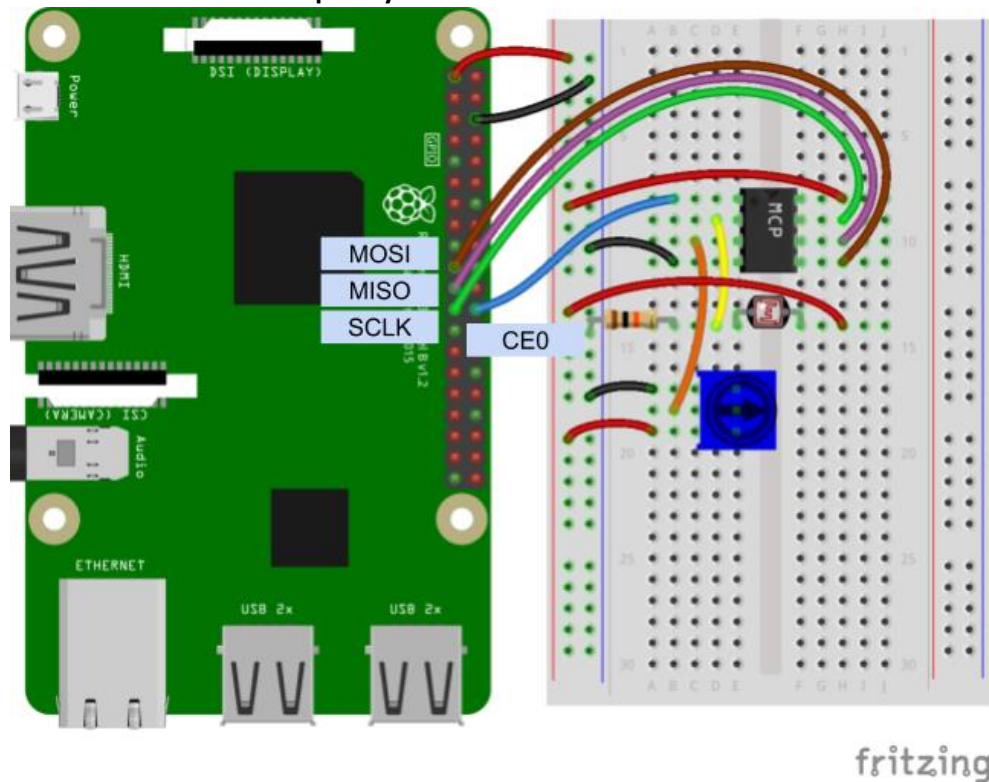
Usaremos el módulo *spidev incorporado* en Python para enviar comandos y leer respuestas en el bus SPI



3.2. Conexiones de hardware

- Conectar **MOSI** (GPIO10, pin 19) a Din en el MCP3002
- Conecte **MISO** (GPIO9, pin 21) a Dout en el MCP3002
- Conecte **SCLK** (GPIO11, pin 21) a CLK en el MCP3002
- Conecte **CE0** (GPIO8, pin 24) a CS / SHDN en el MCP3002
- Conecte el divisor de voltaje de la fotocélula a CH0 en el MCP3002
- Conecte el pin central del potenciómetro a CH1 en el MCP3002
- Realice las conexiones de alimentación (3,3 V) y tierra (GND) como se muestra en el diagrama de Fritzing

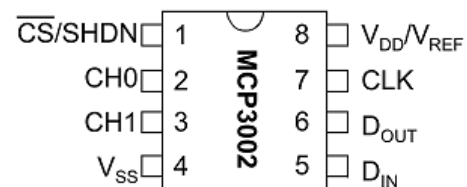
Conexión directa a la Raspberry Pi



Nota: Preste mucha atención a la orientación del MCP3002. Debería ver una muesca en la superficie superior del chip. Con la muesca orientada hacia arriba, la patilla 1 está hacia abajo y a la izquierda de la muesca.

Pinout de la [hoja de datos del MCP3002](#)

MSOP, PDIP, SOIC, TSSOP



3.3. Lectura de voltaje analógico

```
pip install spidev
```

En un archivo nuevo en thonny, ingrese el siguiente código

```
import time
import spidev
spi_ch = 0

# Habilitar SPI
spi = spidev.SpiDev(0, spi_ch)
spi.max_speed_hz = 1200000

def read_adc(adc_ch, vref = 3.3):

    # Asegúrese de que el canal ADC sea 0 o 1
    if adc_ch != 0:
        adc_ch = 1

    # Construir el mensaje SPI
    # Primer bit (Inicio): Lógico alto (1)
    # Segundo bit (SGL / DIFF): 1 para seleccionar el modo simple
    # tercer bit (ODD/SIGN): seleccionar canal (0 or 1)
    # Cuarto bit (MSFB): 0 para LSB primero
    # Sigüientes 12 bits: 0 (No importa)
    msg = 0b11
    msg = ((msg << 1) + adc_ch) << 5
    msg = [msg, 0b00000000]
    reply = spi.xfer2(msg)

    # Construya un entero único a partir de la respuesta (2 bytes)
    adc = 0
    for n in reply:
        adc = (adc << 8) + n

    #El último bit (0) no forma parte del valor de ADC,cambie para eliminarlo
    adc = adc >> 1 # Desplaza de a un bit

    # Calcular el valor de la forma de voltaje ADC
    voltage = (vref * adc) / 1024

    return voltage

#Informe los voltajes del canal 0 y del canal 1 al terminal

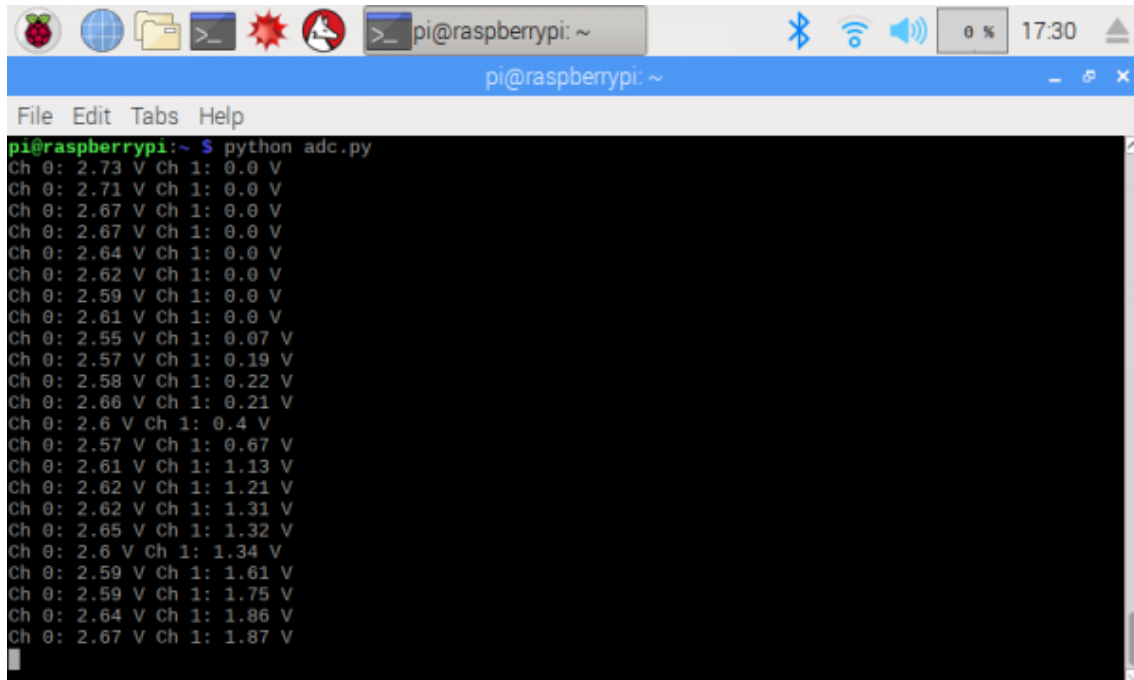
try:
    while True:
        adc_0 = read_adc(0)
        adc_1 = read_adc(1)
        print("Ch 0:", round(adc_0, 2), "V Ch 1:", round(adc_1, 2), "V")
        time.sleep(0.2)

finally:
    GPIO.cleanup()
```

Guarde el archivo (por ejemplo, *adc.py*) y ejecútelo con Python:

```
python adc.py
```

Debería poder cubrir la fotocélula y ver el cambio de voltaje del canal 0. Ajuste la perilla del potenciómetro para ver el cambio de voltaje del canal 1.



```
pi@raspberrypi: ~  
File Edit Tabs Help  
pi@raspberrypi:~$ python adc.py  
Ch 0: 2.73 V Ch 1: 0.0 V  
Ch 0: 2.71 V Ch 1: 0.0 V  
Ch 0: 2.67 V Ch 1: 0.0 V  
Ch 0: 2.67 V Ch 1: 0.0 V  
Ch 0: 2.64 V Ch 1: 0.0 V  
Ch 0: 2.62 V Ch 1: 0.0 V  
Ch 0: 2.59 V Ch 1: 0.0 V  
Ch 0: 2.61 V Ch 1: 0.0 V  
Ch 0: 2.55 V Ch 1: 0.07 V  
Ch 0: 2.57 V Ch 1: 0.19 V  
Ch 0: 2.58 V Ch 1: 0.22 V  
Ch 0: 2.66 V Ch 1: 0.21 V  
Ch 0: 2.6 V Ch 1: 0.4 V  
Ch 0: 2.57 V Ch 1: 0.67 V  
Ch 0: 2.61 V Ch 1: 1.13 V  
Ch 0: 2.62 V Ch 1: 1.21 V  
Ch 0: 2.62 V Ch 1: 1.31 V  
Ch 0: 2.65 V Ch 1: 1.32 V  
Ch 0: 2.6 V Ch 1: 1.34 V  
Ch 0: 2.59 V Ch 1: 1.61 V  
Ch 0: 2.59 V Ch 1: 1.75 V  
Ch 0: 2.64 V Ch 1: 1.86 V  
Ch 0: 2.67 V Ch 1: 1.87 V
```

Código a tener en cuenta:

Estamos usando el canal 0 SPI en la Raspberry Pi cuando inicializamos el objeto SpiDev:

```
spi_ch = 0  
spi = spidev.SpiDev(0, spi_ch)
```

El canal 0 corresponde al uso de CEO (habilitación de chip 0) en los pines de Pi. Si desea utilizar otro dispositivo en el bus SPI, deberá conectarlo a CE1 y utilizar también el canal 1 de SpiDev.

Construimos nuestro mensaje SPI manipulando bits individuales. Comenzamos con el 11 binario (que es el número decimal 3) usando el prefijo '0b':

```
msg = 0b11
```

Si observa la sección 5 en la [hoja de datos](#) del [MCP3002](#), verá que necesitamos enviar un 1 para iniciar la transmisión, seguido de otro 1 para indicar que queremos "modo de un solo extremo". Después de eso, seleccionamos nuestro canal con un 0 (para el canal 0) o un 1 (para el canal 1). Luego, enviamos un 0 para mostrar que queremos que se nos devuelvan los datos con el bit menos significativo (LSB) enviado primero

```
msg = ((msg << 1) + adc_ch) << 5
```

Finalmente, enviamos otros doce ceros. Lo que enviamos aquí realmente no importa, ya que solo necesitamos enviar pulsos de reloj al MCP3002 para que nos envíe datos a través de la línea Dout (MISO). Estos datos (4 bits de configuración seguidos de doce 0) se almacenan en una lista.

```
msg = [msg, 0b00000000]
```

Almacenamos los datos que nos devuelven en la replyvariable, y nos llegan como una lista de 2 bytes (almacenados como 2 enteros). Tenga en cuenta que enviamos datos y leemos la respuesta al mismo tiempo cuando usamos SPI:

```
reply = spi.xfer2(msg)
```

A partir de ahí, construimos un solo entero de los dos bytes (8 bits) desplazando el primero hacia la izquierda en 8 bits y luego agregando el segundo byte. El último bit que leemos es extraño (no forma parte del valor de retorno del ADC), por lo que cambiamos la respuesta a la derecha un bit.

```
adc = 0
for n in reply:
    adc = (adc << 8) + n

adc = adc >> 1
```

El valor de ADC se da como un *porcentaje* del voltaje máximo (cualquiera que sea el voltaje en el pin Vdd / Vref). Ese porcentaje se calcula dividiendo el valor de respuesta por 1024. Obtenemos 1024 porque sabemos que el MCP3002 es un *ADC de 10 bits*, lo que significa que el valor máximo de 10 bits (0b111111111) es 1023. Muchos ADC tienen algún error, por lo que redondee hasta 1024 para facilitar las matemáticas

Una vez que obtenemos el *porcentaje de Vref* con $val / 1024$, multiplicamos ese porcentaje por nuestro *Vref*, que sabemos que es 3.3V en el caso de nuestra Raspberry Pi.

```
voltage = (vref * adc) / 1024
```

¡Y así es como obtenemos nuestra lectura de voltaje analógico! Si todo esto es confuso, simplemente puede copiar la Enable SPIparte y la read_adc()función en su propio código. Luego, simplemente llame read_adc(0)para obtener el voltaje en CH0 en el MCP3002.

Un último fragmento de código interesante es la idea de *parámetros predeterminados*. Si echas un vistazo a la read_adc()definición:

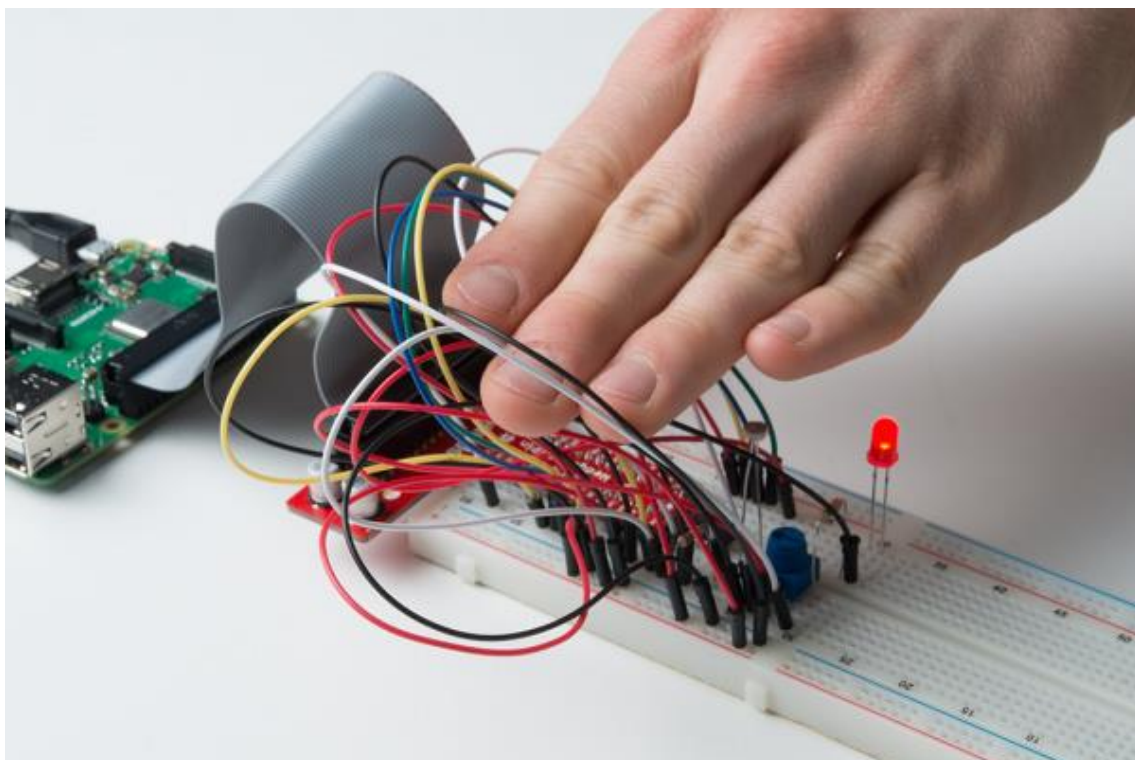
```
def read_adc(adc_ch, vref = 3.3):
```


Verá que en realidad hay dos parámetros: `adc_chy` `vref`. Cuando llama a esta función, debe darle un número de canal (0 o 1). Sin embargo, *opcionalmente* puede enviarle un argumento con el valor `Vref`. En la mayoría de los casos con Raspberry Pi, el voltaje será de 3.3V. Si usa otro voltaje (por ejemplo, 5 V), puede cambiar las matemáticas para que el ADC le brinde una lectura más precisa.

Tiene la opción de llamar a esta función con otro `Vref` (por ejemplo, 5) usando `read_adc(0, 5)` o nombrando explícitamente el `vref` parámetro `read_adc(0, vref=5)`. Sin embargo, debido a que sabemos que hemos conectado 3.3V al MCP3002, simplemente podemos llamar `read_adc(0)` y saber que la función se basará en su parámetro predeterminado `vref=3.3` al hacer sus cálculos.

3.4. Desafío

agregue un LED a su circuito. Escribe un programa que actúe como una luz nocturna variable. Es decir, el LED debe encenderse cada vez que la fotocélula ve oscuridad (poca luz ambiental) y debe apagarse cada vez que la fotocélula ve luz (mucha luz ambiental). Haga que el potenciómetro controle el brillo del LED cuando está encendido. Sugerencia: es posible que desee tomar algunas medidas para determinar el umbral de luz frente a oscuridad. ¿Cuál es el voltaje cuando cubre la fotocélula con la mano?



HAGA CLIC PARA VER LA SOLUCIÓN

Conecte un LED y una resistencia a GPIO12 (pin 32), como en el experimento anterior.

```

import time
import spidev
import RPi.GPIO as GPIO

#Definir pin 12 como led_pin
led_pin = 12

# establecemos limite entre prendido y apagado del Led (Volts)
light_threshold = 2.2

# Usamos GPIO por numero de pines
GPIO.setmode(GPIO.BCM)

# definimos a led_in como Salida
GPIO.setup(led_pin, GPIO.OUT)

# Inicilizamos la salida pwm en 50hz y con un ciclo util de 0%
pwm = GPIO.PWM(led_pin, 50)
pwm.start(0)

# Canal SPI (usa pin CE0)
spi_ch = 0

# Habilitamos SPI
spi = spidev.SpiDev(0, spi_ch)
spi.max_speed_hz = 1200000

def read_adc(adc_ch, vref = 3.3):

    # Asegúrese de que el canal ADC sea 0 o 1
    if adc_ch != 0:
        adc_ch = 1

    # Construimos el mensaje SPI
    # Primer bit (Inicio): Lógico alto (1)
    # Segundo bit (SGL / DIFF): 1 para seleccionar el modo simple
    # tercer bit (ODD/SIGN): seleccionar canal (0 or 1)
    # Cuarto bit (MSFB): 0 para LSB primero
    # Sigüientes 12 bits: 0 (No importa)
    msg = 0b11
    msg = ((msg << 1) + adc_ch) << 5
    msg = [msg, 0b00000000]
    reply = spi.xfer2(msg)

    # Construya un entero único a partir de la respuesta (2 bytes)
    adc = 0
    for n in reply:
        adc = (adc << 8) + n

```

```

#El último bit (0) no forma parte del valor de ADC,cambie para eliminarlo
adc = adc >> 1 # Desplaza de a un bit

# Calcular el valor de la forma de voltaje ADC
voltage = (vref * adc) / 1024

return voltage

# Lea los valores de ADC y determine si el LED debe estar encendido o apagado
try:
    while True:

        # Leemos los valor de ADC
        light_val = read_adc(0)
        knob_val = read_adc(1)

        # Calculamos el brillo para cuando el led esta prendido
        # el brillo máximo es en el 100% del ciclo util
        brightness = (knob_val / 3.3) * 100

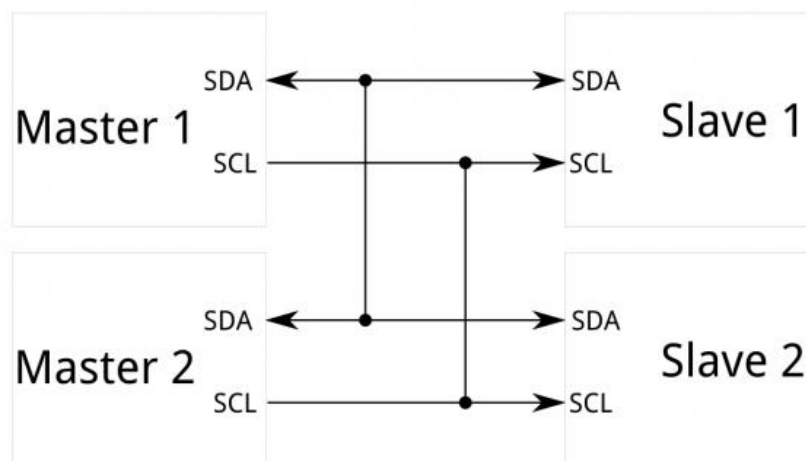
        # Apagamos el LED si la luz ambiental está por encima del umbral
        # Encendemos el LED si la luz ambiental es igual o inferior al umbral
        if light_val > light_threshold:
            pwm.ChangeDutyCycle(0)
        else:
            pwm.ChangeDutyCycle(brightness)
        time.sleep(0.1)

finally:
    pwm.stop()
    GPIO.cleanup()

```

4. I2C

4.1 Introducción

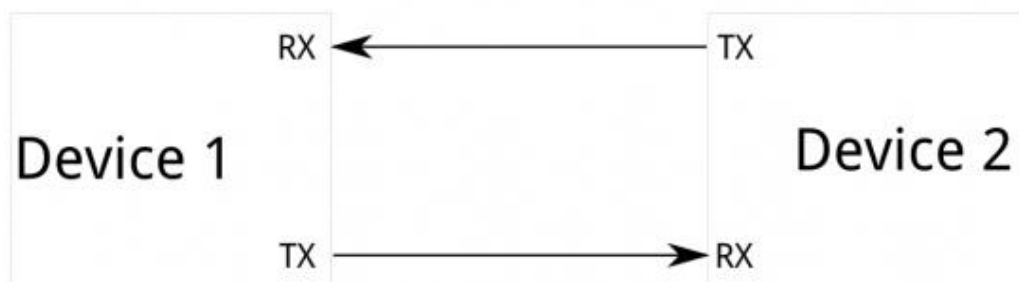


El protocolo de circuito integrado (I^2C) es un protocolo destinado a permitir que varios circuitos integrados digitales "esclavos" ("chips") se comuniquen con uno o más chips "maestros". Al igual que la Interfaz periférica en serie (SPI), solo está diseñada para comunicaciones de corta distancia dentro de un solo dispositivo. Al igual que las interfaces seriales asíncronas (como RS-232 o UART), solo requiere dos cables de señal para intercambiar información.

4.2 ¿Por qué utilizar I2C?

Para averiguar por qué uno podría querer comunicarse a través de I^2C , primero debe compararlo con las otras opciones disponibles para ver en qué se diferencia.

4.3 ¿Qué hay de malo con los puertos UART?



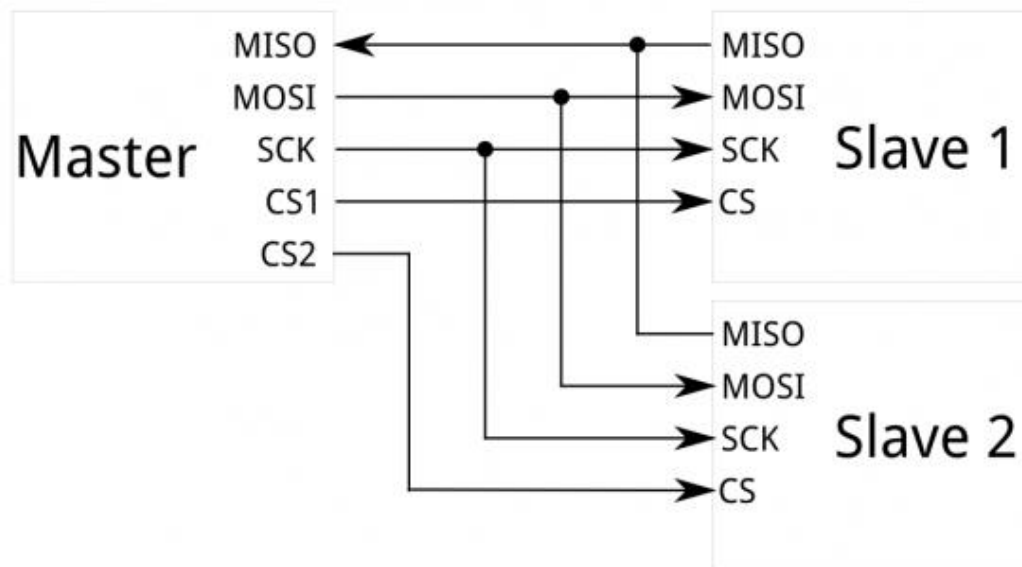
Debido a que los puertos seriales son **asíncronos** (no se transmiten datos de reloj), los dispositivos que los usan deben acordar con anticipación una velocidad de datos. Los dos dispositivos también deben tener relojes que estén cerca de la misma velocidad, y lo seguirán siendo: las diferencias excesivas entre las velocidades de reloj en ambos extremos causarán datos confusos.

Los puertos serie asíncronos requieren una sobrecarga de hardware: el UART en cada extremo es relativamente complejo y difícil de implementar con precisión en el software si es necesario. Al menos un bit de inicio y parada es parte de cada trama de datos, lo que significa que se requieren 10 bits de tiempo de transmisión por cada 8 bits de datos enviados, lo que consume la velocidad de datos.

Otro defecto fundamental de los puertos serie asíncronos es que son inherentemente adecuados para las comunicaciones entre dos, y solo dos, dispositivos. Si bien es *posible* conectar varios dispositivos a un solo puerto serie, [la contención del bus](#) (donde dos dispositivos intentan conducir la misma línea al mismo tiempo) siempre es un problema y debe tratarse con cuidado para evitar daños a los dispositivos en cuestión. generalmente a través de hardware externo.

Finalmente, la velocidad de datos es un problema. Si bien no existe *un* límite *teórico* para las comunicaciones en serie asíncronas, la mayoría de los dispositivos UART solo admiten un determinado conjunto de velocidades en baudios fijas, y la más alta de ellas suele rondar los 230400 bits por segundo.

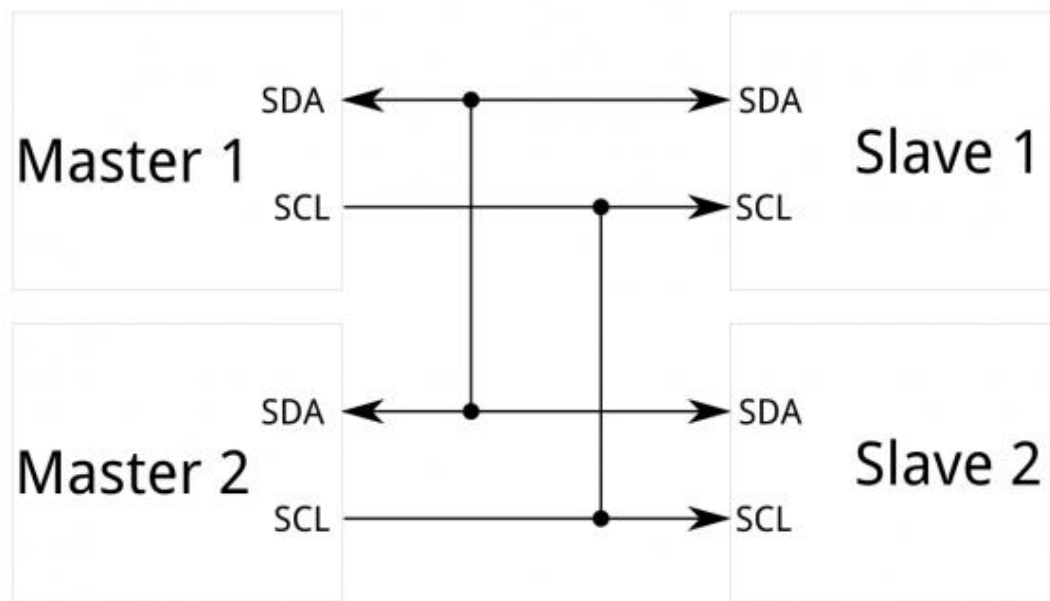
4.4 ¿Qué esta mal con SPI?



El inconveniente más obvio de SPI es la cantidad de pines necesarios. La conexión de un solo maestro a un solo esclavo con un bus SPI requiere cuatro líneas; cada esclavo adicional requiere un pin de E / S de selección de chip adicional en el maestro. La rápida proliferación de conexiones de clavijas lo hace indeseable en situaciones en las que muchos dispositivos deben ser esclavos de un maestro. Además, la gran cantidad de conexiones para cada dispositivo puede dificultar el enrutamiento de señales en situaciones de diseño de PCB estrecho. SPI solo permite un maestro en el bus, pero admite un número arbitrario de esclavos (sujeto solo a la capacidad de accionamiento de los dispositivos conectados al bus y al número de pines de selección de chip disponibles).

SPI es bueno para conexiones **full-duplex** (envío y recepción simultáneos de datos) de alta velocidad de datos, admite velocidades de reloj superiores a 10MHz (y por lo tanto, 10 millones de bits por segundo) para algunos dispositivos, y la velocidad escala muy bien. El hardware en cada extremo suele ser un registro de desplazamiento muy simple, lo que permite una fácil implementación en el software.

4.5 I²C - ¡Lo mejor de ambos mundos!



I²C requiere solo dos cables, como serie asíncrona, pero esos dos cables pueden admitir hasta 1008 dispositivos esclavos. Además, a diferencia de SPI, I²C puede admitir un sistema multimaestro, lo que permite que más de un maestro se comuniquen con todos los dispositivos en el bus (aunque los dispositivos maestros no pueden comunicarse entre sí a través del bus y deben turnarse para usar el líneas de bus).

Las velocidades de datos se encuentran entre la serie asincrónica y la SPI; la mayoría ^{de} los dispositivos I²C pueden comunicarse a 100 kHz o 400 kHz. Hay algo de sobrecarga con I²C; por cada 8 bits de datos que se envíen, se debe transmitir un bit adicional de metadatos (el bit "ACK / NACK", del que hablaremos más adelante).

El hardware necesario para implementar I²C es más complejo que el SPI, pero menos que el serial asíncrono. Puede implementarse de manera bastante trivial en software.

4.6 I²C - Una breve historia

I²C fue desarrollado originalmente en 1982 por Philips para varios chips Philips. La especificación original permitía solo comunicaciones de 100 kHz y solo se proporcionaba para direcciones de 7 bits, lo que limitaba el número de dispositivos en el bus a 112 (hay varias direcciones reservadas, que nunca se utilizarán para direcciones I²C válidas). En 1992, se publicó la primera especificación pública, agregando un modo rápido de 400 kHz y un espacio de direcciones ampliado de 10 bits. La mayor parte del tiempo (por ejemplo, en el dispositivo ATmega328 en muchas placas compatibles con Arduino), el soporte del dispositivo para I²C finaliza en este punto. Hay tres modos adicionales especificados:

- modo rápido plus, a 1MHz
- modo de alta velocidad, a 3.4MHz
- modo ultrarrápido, a 5MHz.

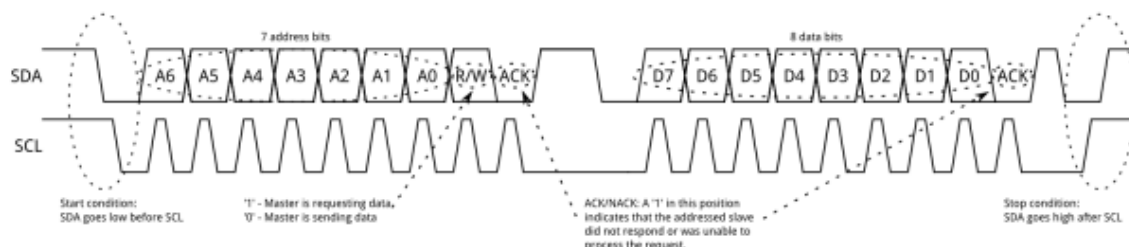
Además de "vainilla" I²C, Intel introdujo una variante en 1995 llamada " **System Management Bus**" (SMBus) . SMBus es un formato más controlado, destinado a maximizar la predictibilidad de las comunicaciones entre los IC de soporte en las placas base de las PC. La diferencia más significativa entre SMBus es que limita las velocidades de 10kHz a 100kHz, mientras que I²C puede admitir dispositivos de 0kHz a 5MHz. SMBus incluye un modo de tiempo de espera de reloj que hace que las operaciones de baja velocidad sean ilegales, aunque muchos dispositivos SMBus lo admitirán de todos modos para maximizar la interoperabilidad con los sistemas I²C integrados.

4.7 Protocolo

La comunicación a través de I²C es más compleja que con una solución UART o SPI. La señalización debe adherirse a un protocolo determinado para que los dispositivos del bus la reconozcan como comunicaciones I²C válidas . Afortunadamente, la mayoría de los dispositivos se encargan de todos los detalles complicados por usted, lo que le permite concentrarse en los datos que desea intercambiar.

Lo esencial

Los mensajes se dividen en dos tipos de tramas: una trama de dirección, donde el maestro indica al esclavo al que se envía el mensaje, y una o más tramas de datos, que son mensajes de datos de 8 bits que se pasan del maestro al esclavo o viceversa. . Los datos se colocan en la línea SDA después de que SCL baja y se muestrean después de que la línea SCL sube. El tiempo entre el borde del reloj y la lectura / escritura de datos está definido por los dispositivos en el bus y variará de un chip a otro.



Haga clic en la imagen para ver más de cerca.

Condición de inicio

Para iniciar la trama de dirección, el dispositivo maestro deja SCL alto y baja SDA. Esto avisa a todos los dispositivos esclavos de que una transmisión está a punto de comenzar. Si dos dispositivos maestros desean tomar posesión del bus al mismo tiempo, el dispositivo que haga bajar SDA primero gana la carrera y obtiene el control del bus. Es posible emitir arranques repetidos, iniciando una nueva secuencia de comunicación sin ceder el control del bus a otros maestros; hablaremos de eso más tarde.

Marco de dirección

La trama de dirección siempre es la primera en cualquier secuencia de comunicación nueva. Para una dirección de 7 bits, la dirección se sincroniza con el bit más significativo (MSB)

primero, seguido de un bit R / W que indica si se trata de una operación de lectura (1) o escritura (0).

El noveno bit de la trama es el bit NACK / ACK. Este es el caso de todos los marcos (datos o dirección). Una vez que se envían los primeros 8 bits de la trama, el dispositivo receptor recibe control sobre SDA. Si el dispositivo receptor no baja la línea SDA antes del noveno pulso de reloj, se puede inferir que el dispositivo receptor no recibió los datos o no supo cómo analizar el mensaje. En ese caso, el intercambio se detiene y depende del maestro del sistema decidir cómo proceder.

Marcos de datos

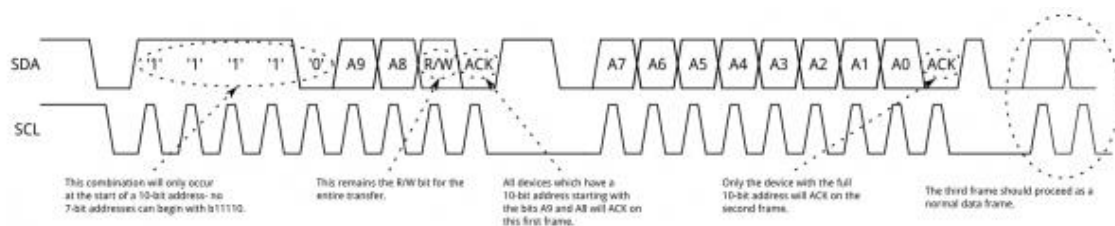
Una vez que se ha enviado la trama de dirección, los datos pueden comenzar a transmitirse. El maestro simplemente continuará generando pulsos de reloj a intervalos regulares, y el maestro o el esclavo colocarán los datos en SDA, dependiendo de si el bit R / W indica una operación de lectura o escritura. El número de tramas de datos es arbitrario y la mayoría de los dispositivos esclavos incrementarán automáticamente el registro interno, lo que significa que las lecturas o escrituras posteriores vendrán del siguiente registro de la línea.

Condición de parada

Una vez que se hayan enviado todos los marcos de datos, el maestro generará una condición de parada. Las condiciones de parada están definidas por una transición 0-> 1 (de baja a alta) en SDA *después de* una transición de 0-> 1 en SCL, con SCL permaneciendo alta. Durante la operación normal de escritura de datos, el valor en SDA **no** debe cambiar cuando SCL es alto, para evitar falsas condiciones de parada.

4.8 Temas de protocolo avanzado

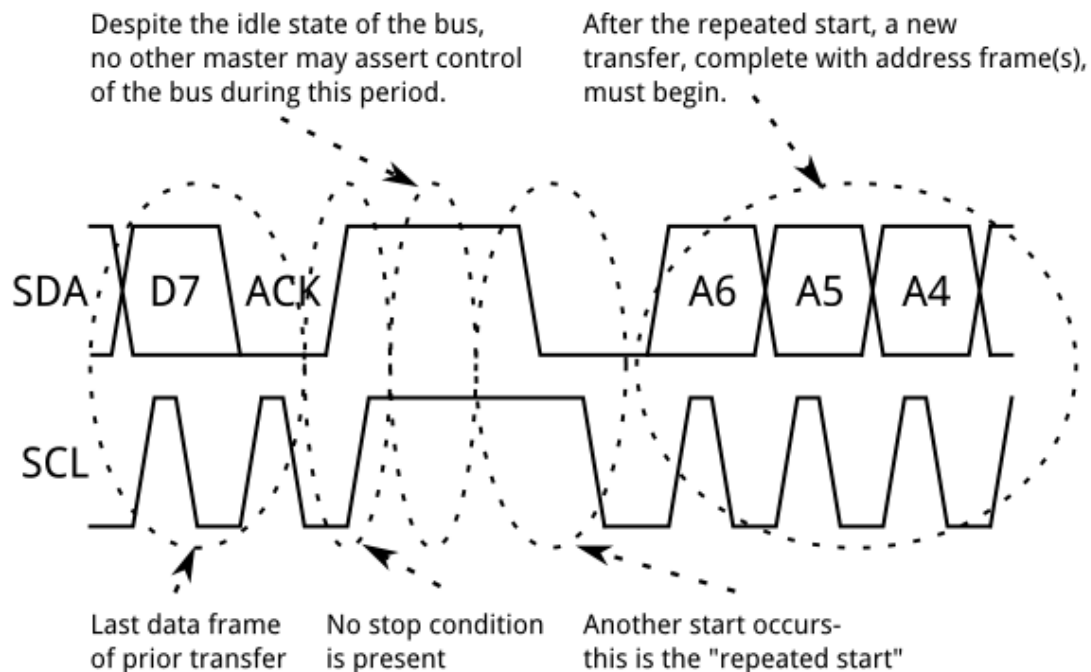
Direcciones de 10 bits



En un sistema de direccionamiento de 10 bits, se requieren dos tramas para transmitir la dirección esclava. La primera trama consistirá en el código b11110xyz, donde 'x' es el MSB de la dirección esclava, y es el bit 8 de la dirección esclava y z es el bit de lectura / escritura como se describió anteriormente. El bit ACK de la primera trama será afirmado por todos los esclavos que coincidan con los dos primeros bits de la dirección. Al igual que con una transferencia normal de 7 bits, otra transferencia comienza inmediatamente y esta transferencia contiene los bits 7: 0 de la dirección. En este punto, el esclavo direccionado debe responder con un bit ACK. Si no es así, el modo de falla es el mismo que el de un sistema de 7 bits.

Tenga en cuenta que los dispositivos de dirección de 10 bits pueden coexistir con los dispositivos de dirección de 7 bits, ya que la parte inicial '11110' de la dirección no forma parte de ninguna dirección válida de 7 bits.

Condiciones de inicio repetidas



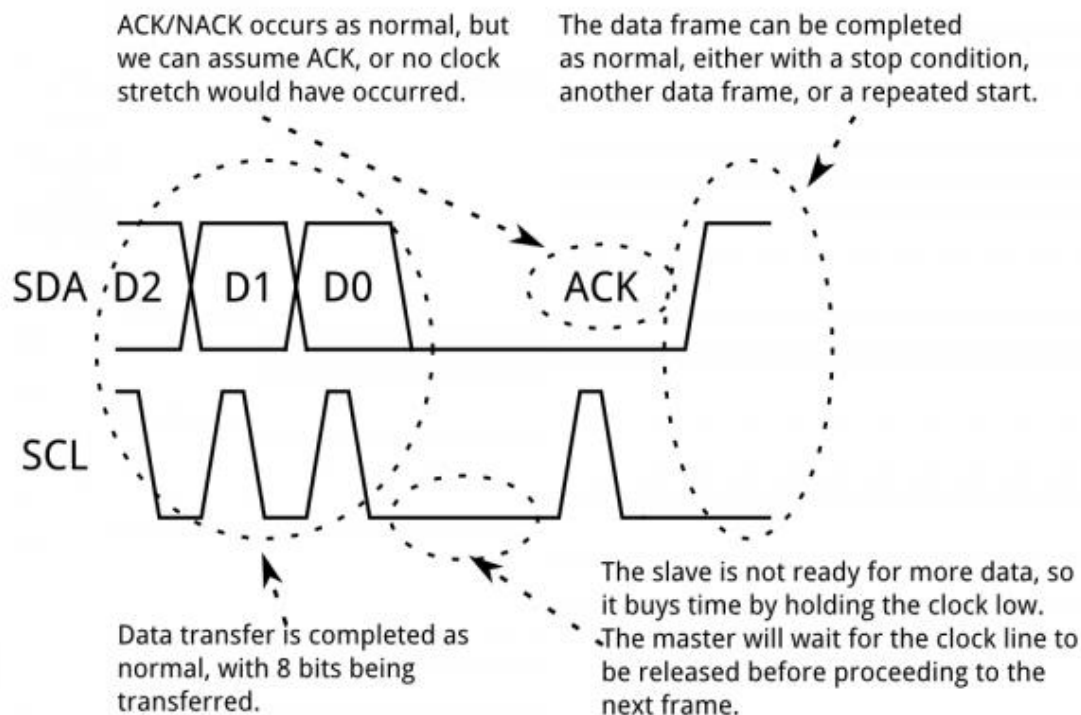
A veces, es importante que un dispositivo maestro pueda intercambiar varios mensajes de una vez, sin permitir que otros dispositivos maestros en el bus interfieran. Por esta razón, se ha definido la condición de inicio repetido.

Para realizar un arranque repetido, se permite que SDA suba mientras que SCL es bajo, SCL se permite que suba y luego SDA se baja nuevamente mientras que SCL es alto. Debido a que no hubo una condición de parada en el bus, la comunicación anterior no se completó realmente y el maestro actual mantiene el control del bus.

En este punto, el siguiente mensaje puede comenzar a transmitirse. La sintaxis de este nuevo mensaje es la misma que la de cualquier otro mensaje: un marco de dirección seguido de marcos de datos. Se permite cualquier número de arranques repetidos y el maestro mantendrá el control del autobús hasta que emita una condición de parada.

Estiramiento del reloj

A veces, la velocidad de datos del maestro excederá la capacidad del esclavo para proporcionar esos datos. Esto puede deberse a que los datos aún no están listos (por ejemplo, el esclavo aún no ha completado una conversión de analógico a digital) o porque una operación anterior aún no se ha completado (por ejemplo, una EEPROM que no se ha completado escribiendo en la memoria no volátil todavía y necesita terminar eso antes de que pueda atender otras solicitudes).

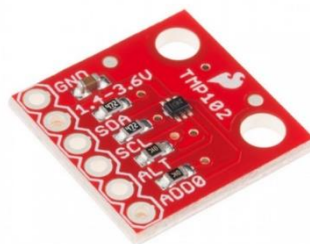


En este caso, algunos dispositivos esclavos ejecutarán lo que se denomina "alargamiento del reloj". Nominalmente, **todo el** reloj es controlado por el dispositivo maestro: los esclavos simplemente colocan datos en el bus o toman datos del bus en respuesta a los pulsos de reloj del maestro. En cualquier punto del proceso de transferencia de datos, un esclavo direccionado puede mantener baja la línea SCL después de que el maestro la libere. Se requiere que el maestro se abstenga de pulsos de reloj adicionales o transferencia de datos hasta el momento en que el esclavo libere la línea SCL.

5.Sensor de temperatura I2C

Además de los sensores analógicos y los chips SPI, a menudo encontrará sensores (y otros dispositivos) que dependen del protocolo Inter-Integrated Chip (IIC o I^2C). Este es un bus de 2 hilos que contiene un reloj y un canal de datos. El maestro (Raspberry Pi) y el dispositivo (sensor) pueden comunicarse en el mismo cable de datos.

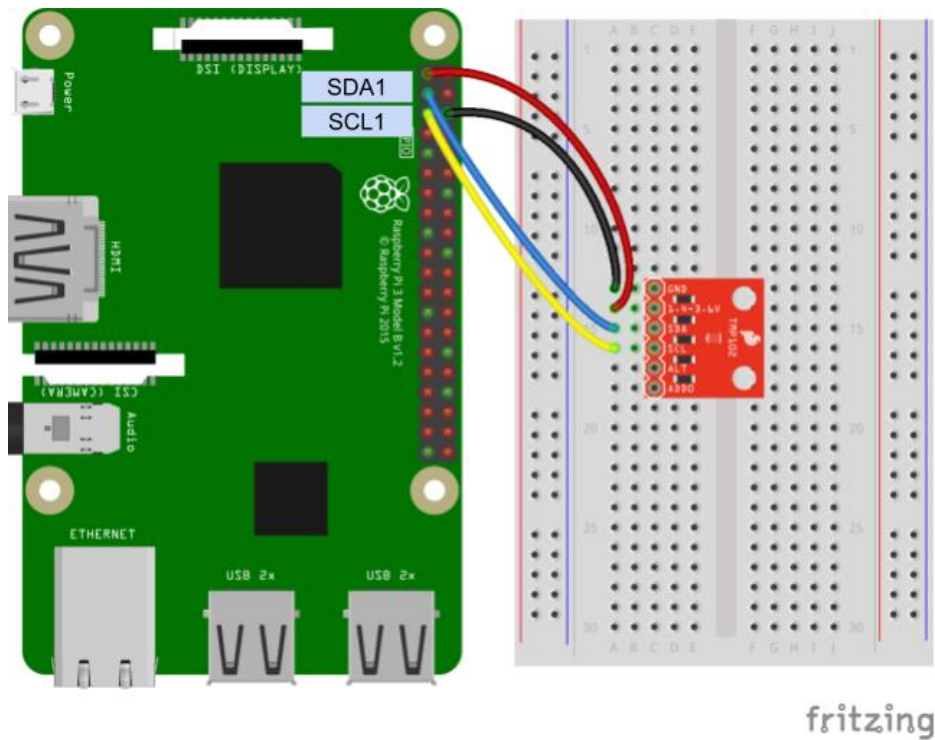
Para ver este protocolo en acción, escribiremos un programa para hablar con un [sensor de temperatura TMP102](#) . Usaremos el módulo *smbus* Python para manejar la comunicación de bajo nivel por nosotros. Tenga en cuenta que "SMBus" significa "System Management Bus" y es otra capa de protocolo construida sobre el protocolo I^2C . Al usar *smbus* , perdemos algunas habilidades de I^2C (por ejemplo, estiramiento del reloj), pero aún podemos hablar con muchos sensores de I^2C .



5.1 Conexiones de hardware

- Conecte **SDA1** (GPIO2, pin 3) a SDA en el TMP102
- Conecte **SCL1** (GPIO3, pin 5) a SCL en el TMP102
- Conecte la alimentación (3,3 V) a VCC en el TMP102
- Conecte la tierra (GND) a GND en el TMP102

Conexión directa a la Raspberry Pi:



5.2 Código: leer y calcular la temperatura

En una terminal, ingrese lo siguiente:

```
sudo apt-get install python3-smbus
```

En un archivo nuevo, copie el siguiente código:

```

import time
import smbus

i2c_ch = 1

# Dirección del TMP102 en el bus I2C
i2c_address = 0x48

# Registro de Direcciones
reg_temp = 0x00
reg_config = 0x01

# Calcular el complemento 2's de un numero
def twos_comp(val, bits):
    if (val & (1 << (bits - 1))) != 0:
        val = val - (1 << bits)
    return val

# Leer los registros de temperatura y convertirlos a Celcius
def read_temp():

    # Leer los registros de temperatura
    val = bus.read_i2c_block_data(i2c_address, reg_temp, 2)
    # NOTA: val[0] = MSB byte 1, val [1] = LSB byte 2
    #print ("!shifted val[0] = ", bin(val[0]), "val[1] = ", bin(val[1]))

    temp_c = (val[0] << 4) | (val[1] >> 4)
    #print (" shifted val[0] = ", bin(val[0] << 4), "val[1] = ", bin(val[1] >
> 4))
    #print (bin(temp_c))

    # Convertir al complemento 2s (la temperaturas pueden ser negativas)
    temp_c = twos_comp(temp_c, 12)

    # Convertir los regidtros a su valor en temperatura (C)
    temp_c = temp_c * 0.0625

    return temp_c

# Inicializar I2C (SMBus)
bus = smbus.SMBus(i2c_ch)

# Leer la CONFIGURACION del registro (2 bytes)
val = bus.read_i2c_block_data(i2c_address, reg_config, 2)
print("Old CONFIG:", val)

# Establecer en muestreo de 4 Hz (CR1, CR0 = 0b10)
val[1] = val[1] & 0b00111111

```

```

val[1] = val[1] | (0b10 << 6)

# Escribe el muestreo de 4 Hz de nuevo en la CONFIGURACION
bus.write_i2c_block_data(i2c_address, reg_config, val)

# Leer CONFIGURACION para verificar que lo cambiamos.
val = bus.read_i2c_block_data(i2c_address, reg_config, 2)
print("New CONFIG:", val)

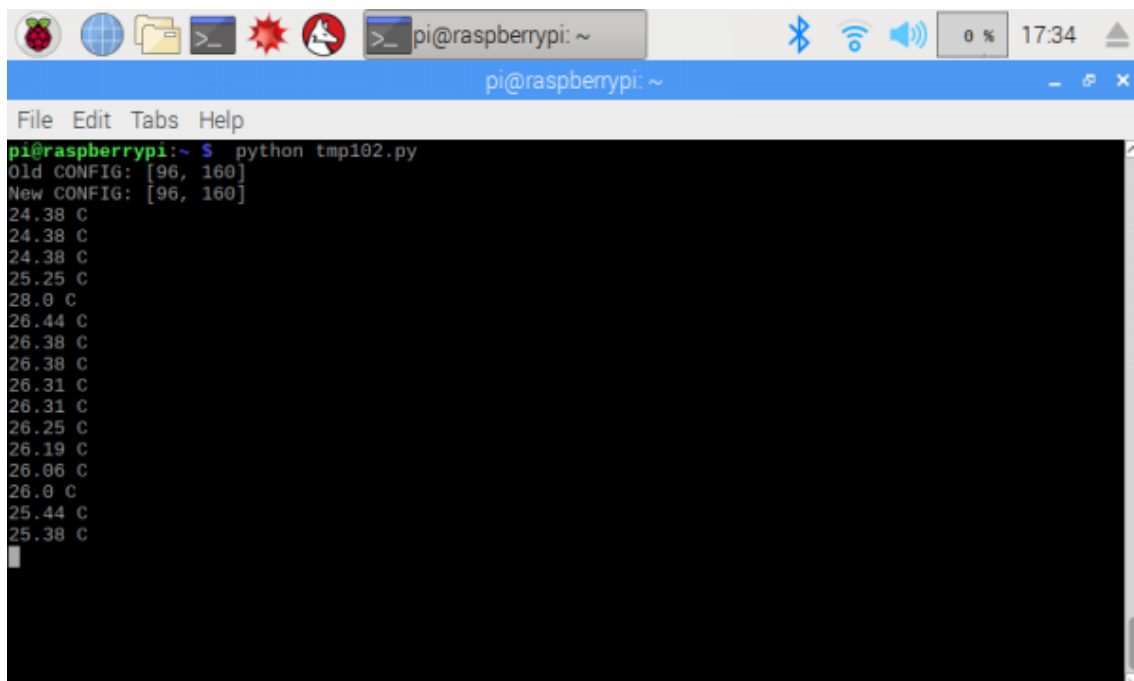
# Imprime la temperatura cada segundo
while True:
    temperature = read_temp()
    print(round(temperature, 2), "C")
    time.sleep(1)

```

Guarde el archivo (por ejemplo, *tmp102.py*) y ejecútelo con Python:

```
python tmp102.py
```

Debería ver que los 2 bytes en el registro CONFIG se actualizan y luego la temperatura se imprime en la pantalla cada segundo.



```

pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ python tmp102.py
Old CONFIG: [96, 160]
New CONFIG: [96, 160]
24.38 C
24.38 C
24.38 C
25.25 C
28.0 C
26.44 C
26.38 C
26.38 C
26.31 C
26.31 C
26.25 C
26.19 C
26.06 C
26.0 C
25.44 C
25.38 C

```

Código a tener en cuenta:

A diferencia de SPI, I²C se basa en un conjunto de direcciones y registros. Esto se debe a que SPI está configurado para hablar con un chip a la vez, mientras que I²C puede compartir muchos dispositivos en un bus. Para evitar conflictos, las direcciones se asignan a los dispositivos (por el fabricante) para que cada uno sepa cuándo el host (el Pi) está tratando de hablar con él. La dirección de nuestro TMP102 es 0x48.

Cada vez que queremos hablar con el TMP102, debemos enviar su dirección (0x48) en el bus. Solo entonces podemos enviar la ubicación de memoria (o *dirección*) del *registro desde el* que queremos leer o escribir en el TMP102. Tenga en cuenta que para la mayoría^{de} los dispositivos I² C, un *registro* es una ubicación en la memoria del dispositivo que almacena 8 bits (1 byte) de datos. A veces, estos datos controlan la función del dispositivo (como en el caso del registro CONFIG). Otras veces, los registros contienen los datos de lectura del sensor (como en el caso del registro de temperatura).

Usamos el siguiente comando para leer 2 bytes del registro de temperatura en el TMP102:

```
val = bus.read_i2c_block_data(i2c_address, reg_temp, 2)
```

Estos valores se almacenan como una lista [x, y] en la variable. Al mirar la [hoja de datos](#) del [TMP102](#), vemos que la temperatura es de 12 bits. Cuando leemos los dos bytes que contienen esta lectura, necesitamos eliminar los últimos 4 bits del segundo byte. También movemos el primer byte sobre 4 bits:

```
temp_c = (val[0] << 4) | (val[1] >> 4)
```

Para mostrar números negativos para la temperatura, los valores del TMP102 pueden venir en forma de *complemento a dos*. En esto, el primer bit del número de 12 bits determina si el valor es positivo o negativo (0 para positivo, 1 para negativo).

Para convertir un número de complemento a dos en un número negativo en Python, verificamos si el primer bit es 0 o 1. Si es 0, entonces usamos el número tal cual (es positivo!). Si es un 1, restamos el número máximo negativo del complemento a dos ($2^{12} = 4096$ en este caso) de nuestro número.

```
if (val & (1 << (bits - 1))) != 0:  
    val = val - (1 << bits)
```

5.2 Desafío

cambie el registro CONFIG para que el TMP102 actualice su lectura de temperatura 8 veces por segundo (en lugar de 8). Además, imprima la temperatura en grados Fahrenheit. Sugerencia: consulte la página 7 de la [hoja de datos](#) del [TMP102](#) para ver qué bits deben cambiarse en el registro CONFIG.

HAGA CLIC PARA VER LA SOLUCIÓN

```

import time
import smbus

i2c_ch = 1

# TMP102 address on the I2C bus
i2c_address = 0x48

# Register addresses
reg_temp = 0x00
reg_config = 0x01

# Calculate the 2's complement of a number
def twos_comp(val, bits):
    if (val & (1 << (bits - 1))) != 0:
        val = val - (1 << bits)
    return val

# Read temperature registers and calculate Celsius
def read_temp():

    # Read temperature registers
    val = bus.read_i2c_block_data(i2c_address, reg_temp, 2)
    # NOTE: val[0] = MSB byte 1, val [1] = LSB byte 2
    #print ("!shifted val[0] = ", bin(val[0]), "val[1] = ", bin(val[1]))

    temp_c = (val[0] << 4) | (val[1] >> 4)
    #print (" shifted val[0] = ", bin(val[0] << 4), "val[1] = ", bin(val[1] >
> 4))
    #print (bin(temp_c))

    # Convert to 2s complement (temperatures can be negative)
    temp_c = twos_comp(temp_c, 12)

    # Convert registers value to temperature (C)
    temp_c = temp_c * 0.0625

    return temp_c

# Initialize I2C (SMBus)
bus = smbus.SMBus(i2c_ch)

# Read the CONFIG register (2 bytes)
val = bus.read_i2c_block_data(i2c_address, reg_config, 2)
print("Old CONFIG:", val)

# Set to 8 Hz sampling (CR1, CR0 = 0b11)

```

```

val[1] = val[1] & 0b00111111
val[1] = val[1] | (0b11 << 6)

# Write 8 Hz sampling back to CONFIG
bus.write_i2c_block_data(i2c_address, reg_config, val)

# Read CONFIG to verify that we changed it
val = bus.read_i2c_block_data(i2c_address, reg_config, 2)
print("New CONFIG:", val)

# Print out temperature every second
while True:
    temperature = read_temp()
    temp_f = temperature * (9 / 5) + 32
    print(round(temp_f, 2), "F")
    time.sleep(1)

```

6.conectar Arduino y Raspberry Pi por comunicación I2C:

En este pequeño tutorial veremos cómo conectar una tarjeta Arduino a una tarjeta Raspberry Pi por comunicación I2C, encontrarás códigos de programación para tus primeras pruebas.

La comunicación I2C se implementa con 2 señales digitales, la señal de datos «SDA» y la señal de reloj «SCL». Tanto la tarjeta Arduino Uno como las tarjetas Raspberry Pi cuentan con hardware integrado para implementar el protocolo. En el caso del Arduino, encontramos la señal SDA en el pin A5 y la señal SCL en el pin A4, y en las Raspberry Pi podemos encontrar las señales SDA y SCL en los GPIO2 y GPIO3 respectivamente.

El ejercicio que realizaremos en este tutorial consiste en controlar el LED integrado al pin #13 del Arduino desde la Raspberry Pi. Para ello ejecutaremos un script de python3 que solicite al usuario insertar un comando para ser enviado al Arduino a través del protocolo I2C, éste a su vez, estará programado para recibir todos los comandos respondiendo solo a 2: «on» para encender el LED y «off» para apagar el LED.

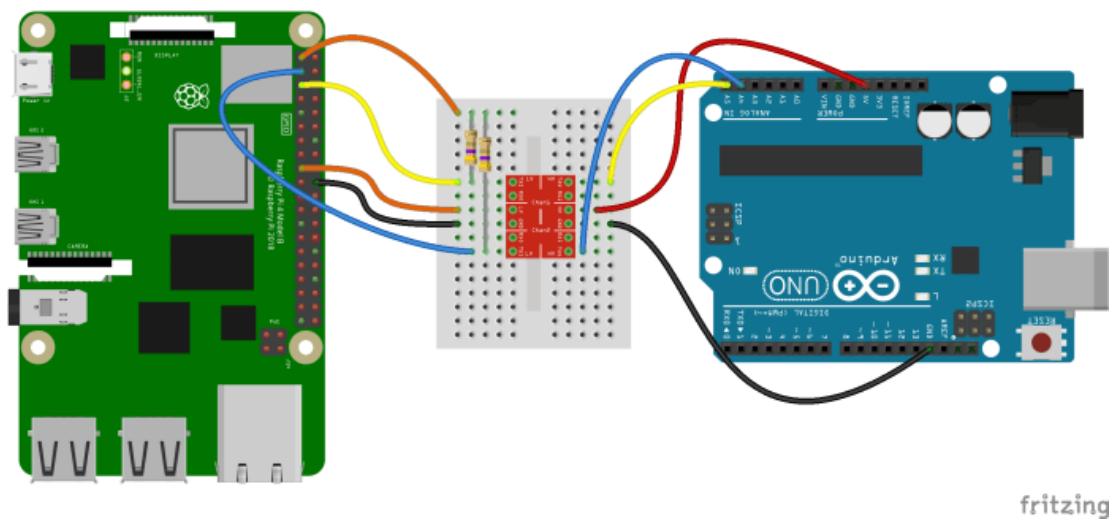
Componentes necesarios

- Raspberry Pi, cualquier modelo
- Arduino UNO
- 2 Resistores de 4.7kohms cada uno

- Protoboard mini o chica
- Cables de conexion M-M y H-M
- Convertidor de nivel lógico bidireccional – 3.3V <-> 5V TTL

Conexiones

Utilizamos un convertidor de nivel lógico para conectar las señales de ambas tarjetas, esto es importante porque trabajan con distintos niveles lógicos de voltaje, el Arduino tienen un nivel lógico de 5V y la Raspberry Pi de 3.3V. Conectar las señales directamente podría resultar en un daño a la tarjeta con nivel lógico menor, es decir, la Raspberry Pi.



Raspberry Pi – Arduino Uno – I2C

Código de Arduino – Arduino Uno

Cargamos el siguiente código a nuestra tarjeta Arduino Uno:

```

#include <Wire.h>

String inputString = "";          // Cadena de texto para
guardar la cadena recibida por I2C

bool stringComplete = false;      // Bandera booleana

int led = 13;

int direccion=31; //Dirección asignada al ARduino. Asegúrate
de que utilices esta dirección en el código de la Raspberry
Pi

void setup() {

    pinMode(led, OUTPUT);

    Wire.begin(direccion);        // Nos unimos al bus i2c bus
con la dirección asignada

    Wire.onReceive(receiveEvent); // Registramos el evento de
recepción de datos

    Serial.begin(9600);

    inputString.reserve(200);

}

void loop() {

    if (stringComplete) {

        Serial.println("Cadena recibida completa");
    }
}

```

```
inputString.remove(0,1); //Removemos 1 carácter. El caracter  
con index 0, caracter adicional sin utilidad que está  
mandando Raspberry Pi
```

```
    if (inputString.equals("off") ){  
        digitalWrite(13,0);  
        Serial.println("Apagado");  
    }  
    else if (inputString.equals("on") ){  
        digitalWrite(13,1);  
        Serial.println("Encendido");  
    }  
  
    Serial.println(" ");  
    inputString = "";  
    stringComplete = false;  
}  
  
}  
  
void receiveEvent(int howMany) {
```

```

while(Wire.available()>0) // Mientras tengamos caracteres en
el buffer

{

    char inChar = (char)Wire.read();

    Serial.print(inChar);

    if (inChar == '\n') {

        stringComplete = true;

    }

    else{

        inputString += inChar;

    }

}

}

```

1.- Habilitar comunicación I2C en la Raspberry Pi

Para que podamos ejecutar un programa que intercambie información a través del protocolo I2C en nuestra Raspberry Pi, debemos habilitar el hardware correspondiente en dicha tarjeta. (Word aparte) Una vez que termines regresa para continuar con este tutorial.

2.- Instalar paquete de Python3

En este código hacemos uso del paquete de Python3 «smbus2». Para instalarlo ejecuta el siguiente comando en la terminal:

```
pi@raspberrypi:~$ sudo pip3 install smbus2
```

Código de Python3 – Raspberry Pi

Copiamos el siguiente código a un script de python en nuestra Raspberry Pi y lo ejecutamos con Python3. Al ejecutarlo, el programa solicitará ingresar un comando, el

comando será enviado al Arduino al oprimir la tecla «Enter». Con el comando «on» se enciende el led integrado del Arduino y con el comando «off» se apaga.

```
from smbus2 import SMBus

import time

try:

    while True:

        with SMBus(1) as bus:

            comando = input("Ingresar comando (on/off): ")

            comando = comando + "\n"

            comando = comando.encode()

            bus.write_i2c_block_data(31, 0, comando)

            time.sleep(0.1)

except KeyboardInterrupt:

    print("\nInterrupcion por teclado")

except ValueError as ve:

    print(ve)

    print("Otra interrupcion")
```