


# Semana 5.

# Sensores y Actuadores.

## Parte 2

SISTEMAS EMBEBIDOS   
(EMBEDED SYSTEMS)

Grado Dual en Industria Digital

Campus Vitoria

Curso 2020-2021



# Deusto

Facultad de Ingeniería  
Ingeniaritza Fakultatea

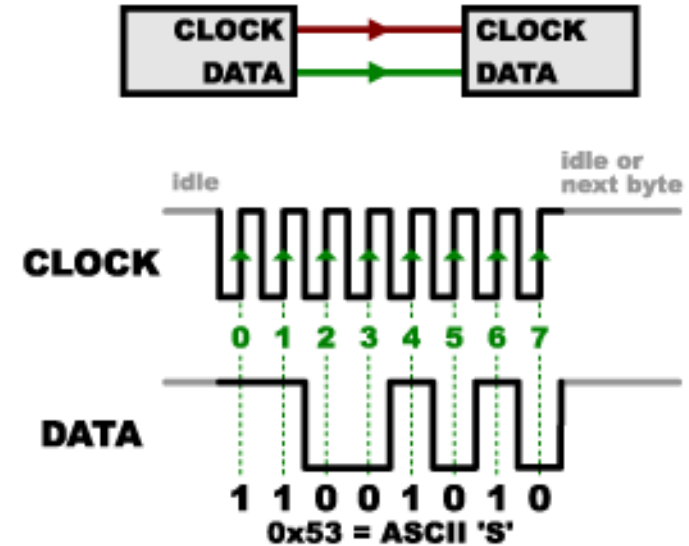
# ¿Cómo proporciona la salida el sensor?

## **RESUMEN**

- *Digital*
- *Analógico (uso de PWM)*
- *En serie/Paralelo*
- *Protocolo comunicación Sincrono (**I2C** y **SPI**)*
- *Protocolo comunicación asíncrono (UART)*

# SPI – Serial Peripheral interface

- Bus de datos síncrono
- Usa líneas separadas para datos y reloj
- Línea de reloj: sincronizar transmisor y receptor
- Señal de reloj: pulso cuadrado que le dice al receptor exactamente cuando tomar las muestras de bits de la línea de datos
- Debido a que la señal de reloj se envía junto con los datos, no es importante especificar previamente la velocidad

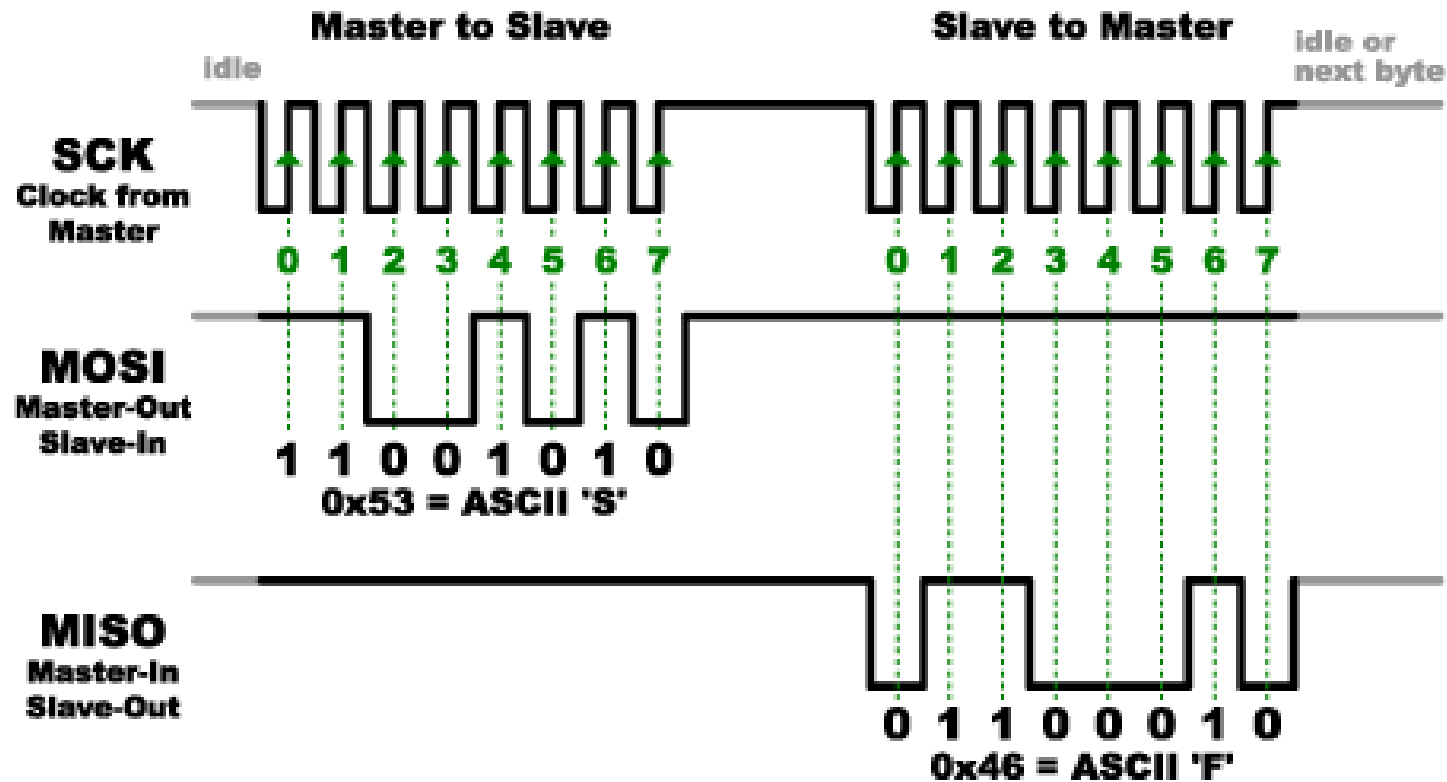
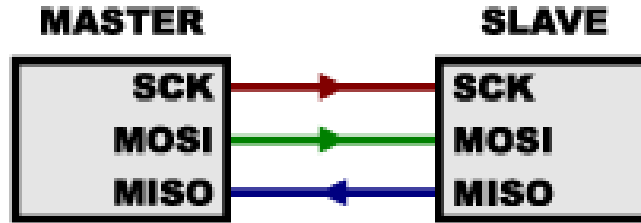


# SPI - Recibir datos

- Solo un extremo genera la señal de reloj (CLK).
- Master: dispositivo que genera la señal. El otro extremo, se llama esclavo (slave)
- Sólo un master simultáneo
- Cuando la información se envía **master** -> **esclavo**: se usa la línea MOSI "Master Out / Slave In".
- Cuando la información se envía **esclavo** -> **master**: se usa la línea MISO "Master In / Slave Out"
- Cuando el esclavo necesita responder al master, el master seguirá enviando la señal de reloj para mantener la sincronización

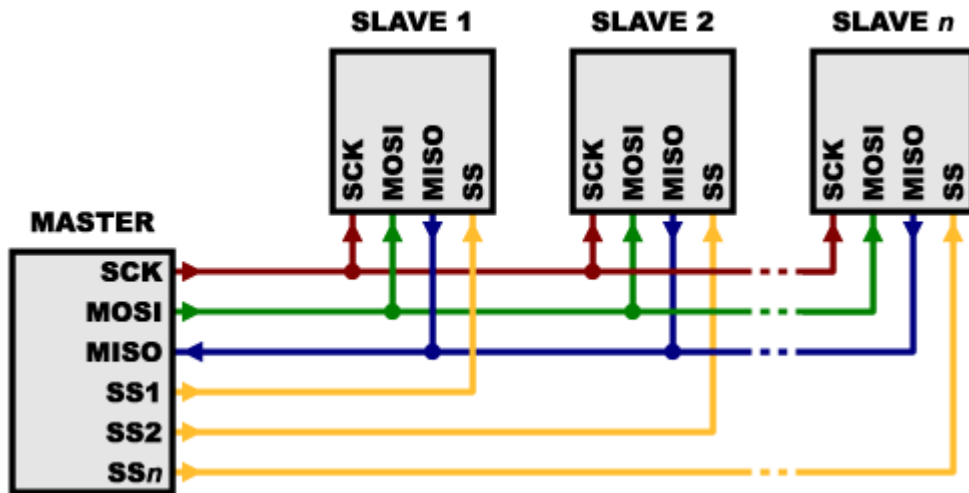
# SPI- Recibir datos

(MOSI) Master Out / Slave In  
(MISO) Master In / Slave Out

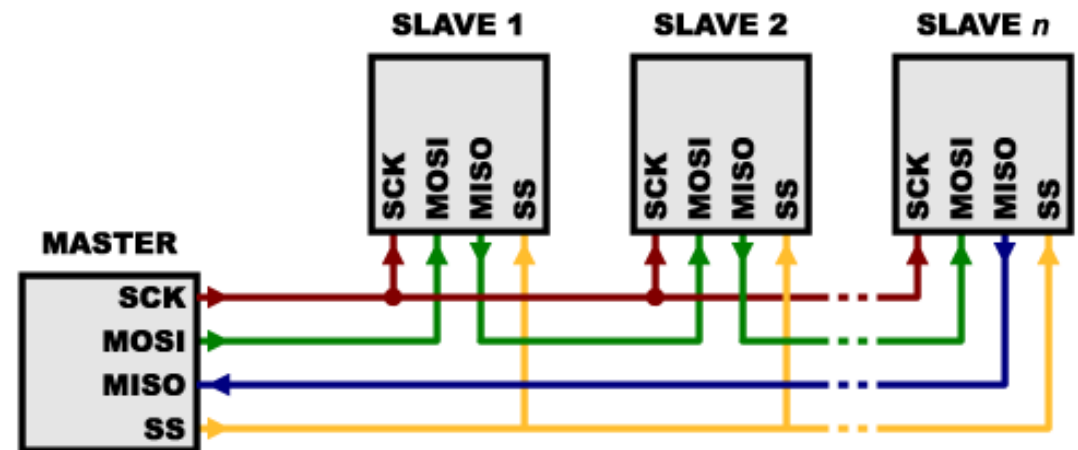


# SPI. Múltiples Esclavos (slaves)

each slave will need a separate SS line.

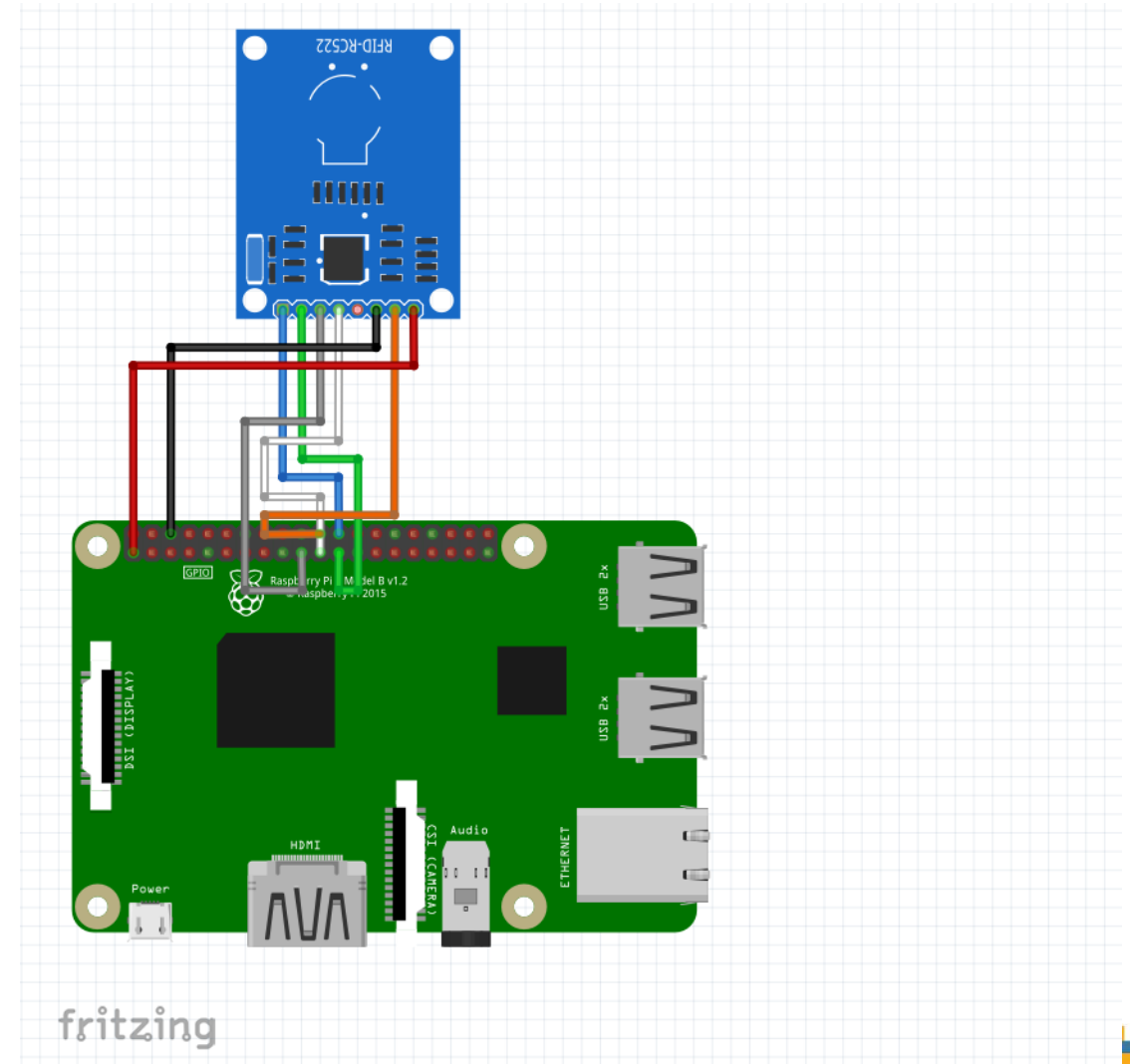
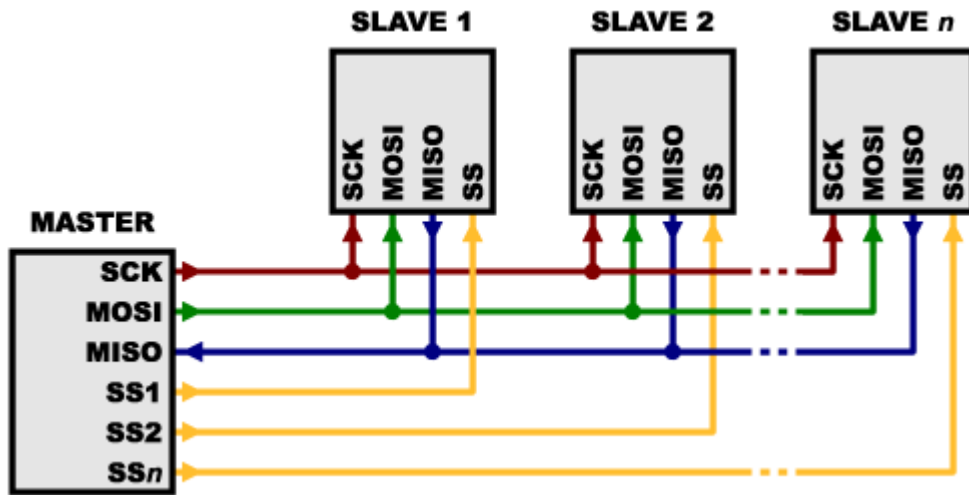


single SS line goes to *all* the slaves



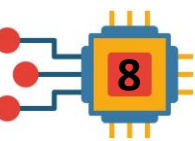
# SPI. Múltiples Esclavos (slaves)

each slave will need a separate SS line.



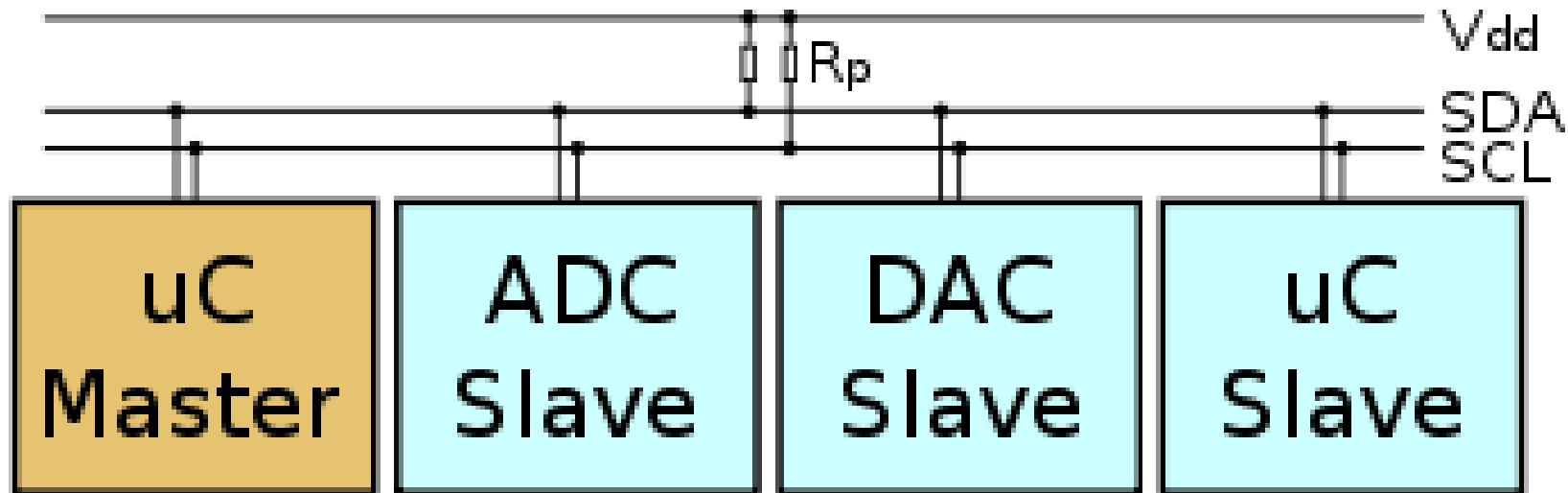
# I2C (inter-Integrated Circuit)

- Bus de comunicaciones serie **sincrona muy utilizado en la industria para la comunicación entre  $\mu$ procesadores y sus perifericos en sistemas integrados.**
- Utiliza únicamente dos líneas para transmitir los datos
  - SDA: Serial Data Line
  - SCL: Serial Clock Line
- Los dispositivos de un bus I2C tienen una dirección única para cada uno, y pueden clasificarse como maestros o como esclavos (Half-duplex).
- Todos los datos están formados por 8 bits y la transmisión comienza con el bit de mas peso.





# I2C (inter-Integrated Circuit)

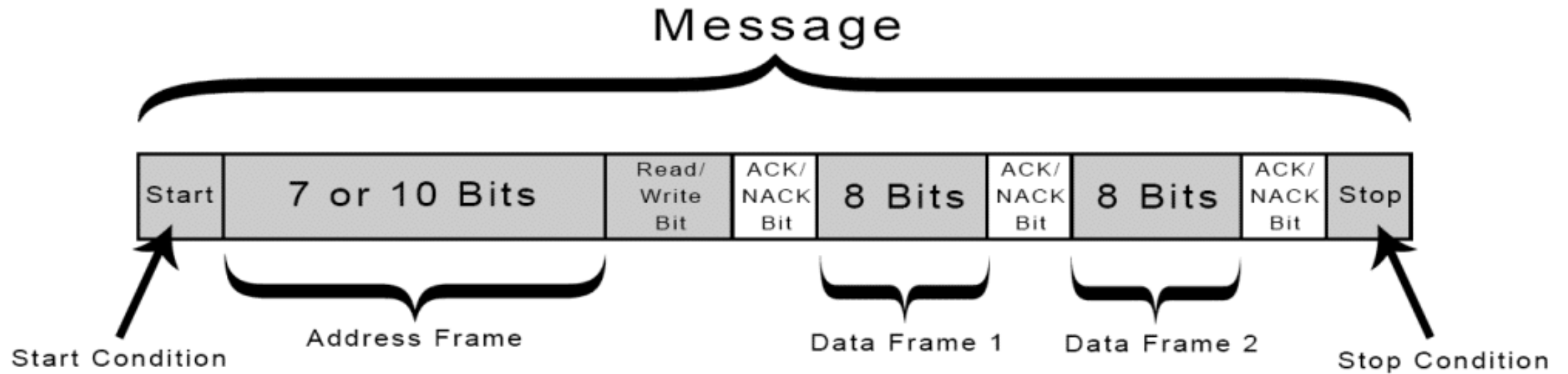


Acelerómetro

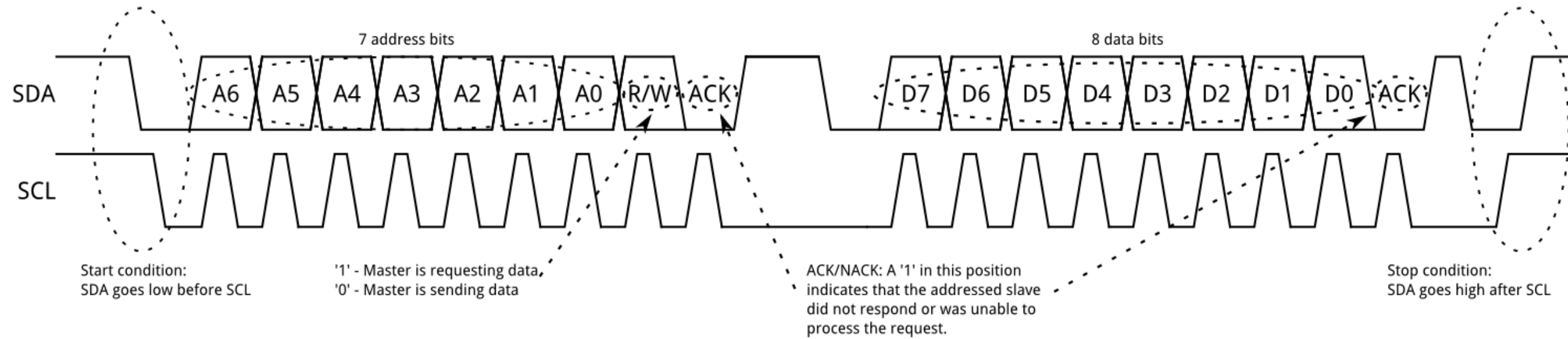


Sensor de Luz

# I2C



# I2C



# ACTIVIDAD DE CLASE

TERMINAREMOS EN CLASE EL DÍA JUEVES 22 DE OCTUBRE

# Formas de romper la RaspberryPi

Nunca tocar la placa con las manos mientras esté alimentada: riesgo de CORTOCIRCUITO

No la desenchufes directamente, mejor cierra ordenadamente el SO (shutdown now)

No pongas la placa encima de superficies metálicas, así evitaremos un cortocircuito. Usa patas de goma o mejor una carcasa

No conectes circuitos que drenen o aporten mucha corriente por las GPIO, el máximo es 2 2-3 mA (en comparación Arduino permite hasta 40 mA, pudiendo alimentar circuitos por las GPIO, RPi no).  
GPIO máximo 3.3V

Siempre repasa 2 o 3 veces la numeración de los pines GPIO: el uso de un PIN incorrecto puede quemar la placa

Mucho cuidado cuando trabajes en modo superusuario/root ya que puedes desconfigurar el sistema

Ante cualquier duda pide ayuda al profesor o consulta Internet en más de una fuente para verificar que la solución es correcta.

Extra-cuidado: los problemas de software son reversibles, los de hardware no.

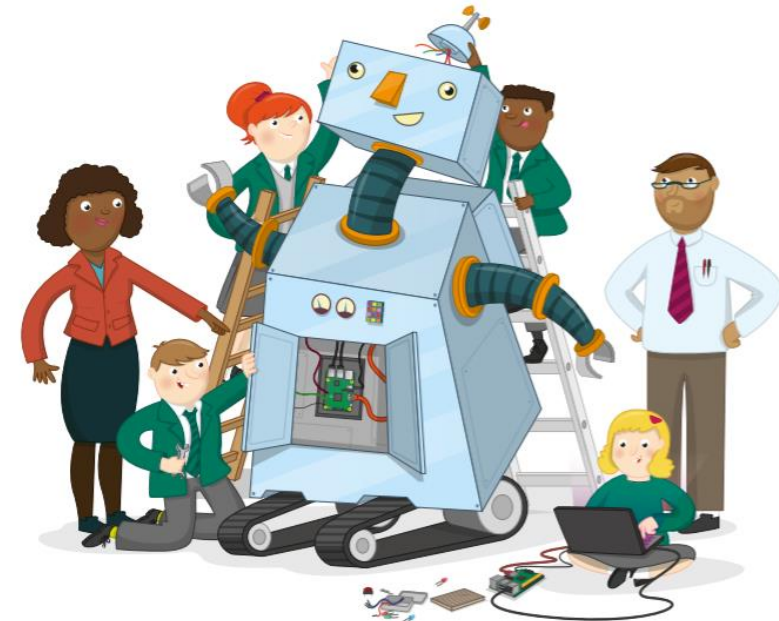


Illustration: [www.raspberrypi.org](http://www.raspberrypi.org)

¿Os habéis preguntado para qué sirve el pin NC – Not connected (sin conexión) ??

On the DHT11 it's connected to pin 6 of the SOIC14 inside.

So it's at least connected to something. But who knows what it's for.

It has a (internal) pull-up resistor to VCC of around 100kOhm on this pin.

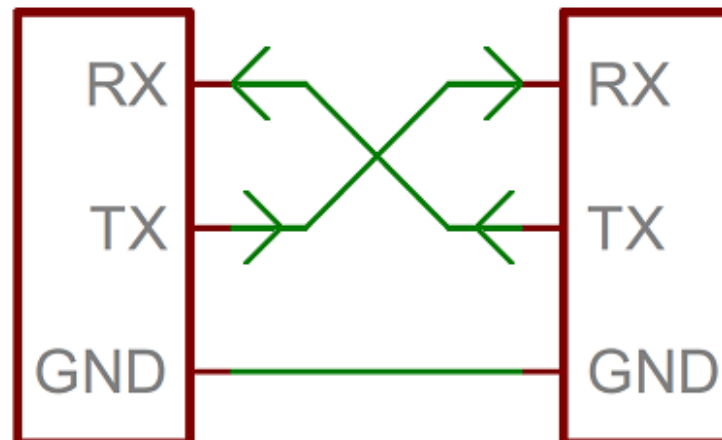
I connected my logic analyzer to it, there is no signal, even while reading the sensor.

Also connecting the Rpi's signal wire to this (incorrect) pin does nothing.

**Factory calibration or testing** would be my guess too, but who knows.

# UART

- **UART (Universal Asynchronous Receiver/Transmitter)** es uno de los protocolos serie mas utilizados.
- Usa una línea de datos simple para transmitir y otra para recibir datos.
- Comúnmente, 8 bits de datos son transmitidos de la siguiente forma: un bit de inicio, a nivel bajo, 8 bits de datos y un bit de parada a nivel alto.



# ¿Cómo proporciona la salida el sensor?

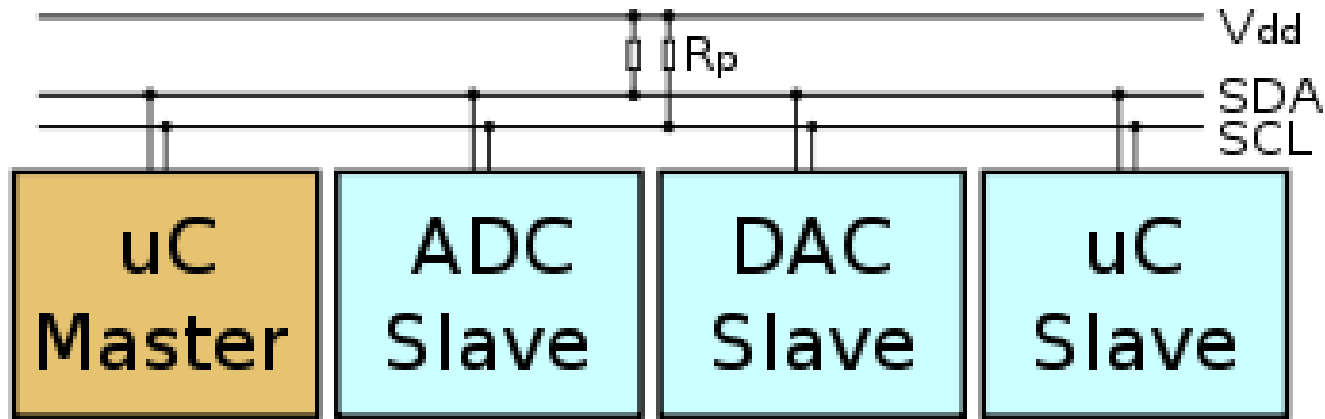
## **RESUMEN**

- *Digital*
- *Analógico (uso de PWM)*
- *En serie/Paralelo*
- *Protocolo comunicación Sincrono (**I2C** y **SPI**)*
- *Protocolo comunicación asíncrono (UART)*



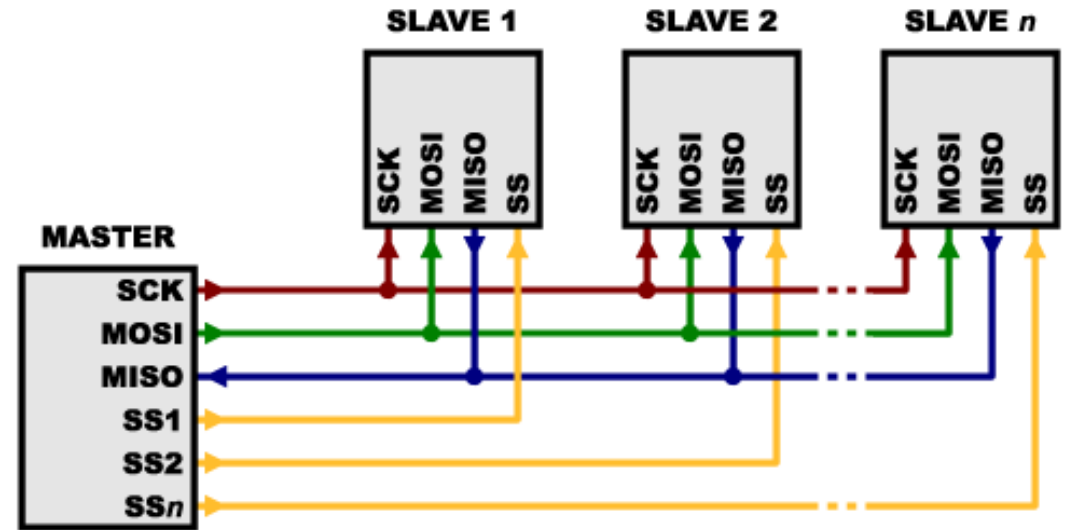
# RESUMEN PROTOCOLOS SERIE

## I2C



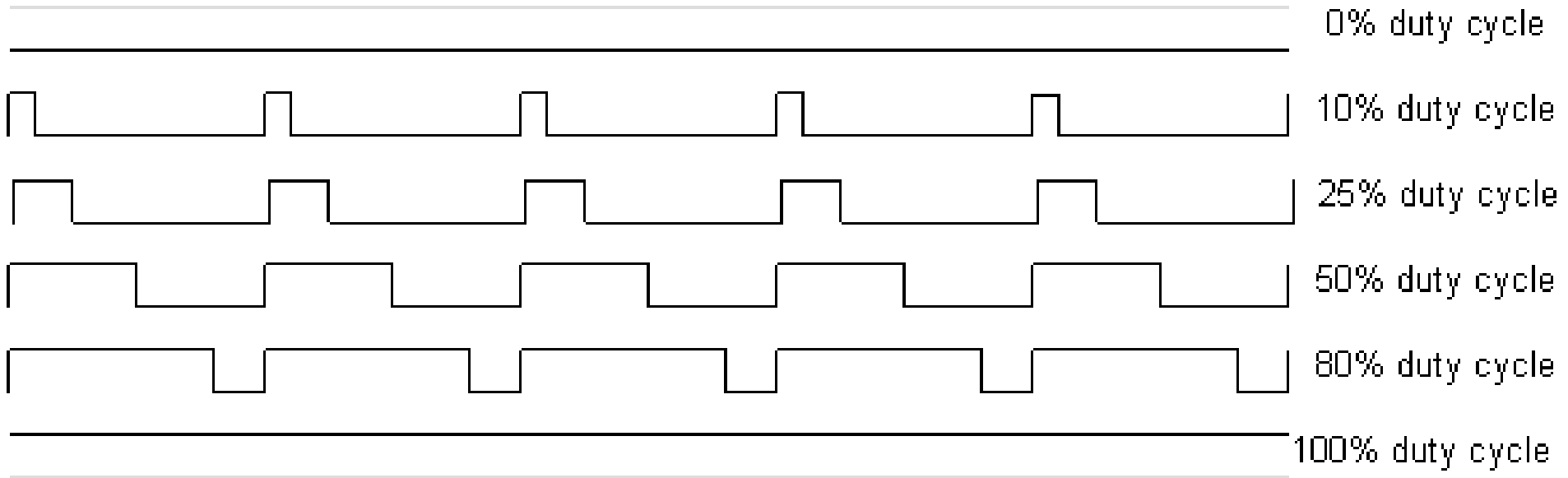
- Comunicación en serie, síncrona
- Half duplex
- 2 líneas de datos (SDA,SCL) (datos, reloj)
- Tramas de datos de 8 bits
- Soporta multiples maestros y esclavos

## SPI



- Comunicación serie asíncrona
- Unico master
- Al menos 4 líneas de datos
  - MOSI: envío **master** -> **esclavo**
  - MISO: envío **esclavo** -> **master**
  - SCK (reloj)
  - SS (selección de esclavo)
- It supports multiple slaves – full duplex
- sincrono

# PWM



PWM es una señal digital cuadrada, donde la **frecuencia es constante**, pero el porcentaje de tiempo que esta en valor alto y bajo (duty cycle) se puede variar en 0 y 100%

# PWM – GPIOs con python

- Paso 1: Crear un generador de pulsos

```
pin_pwm = GPIO.PWM(numcanal, frecuenciaHz)  
pin_pwm = GPIO.PWM(18, 100)
```

- Paso 2: Activar el generador a un nivel (duty cycle, intensidad resultante) entre 0 y 100:

```
pin_pwm.start(70)
```

- Paso 3: Modificar el nivel:

```
pin_pwm.ChangeDutyCycle(24)
```

- Paso 4: Detener:

```
pin_pwm.stop()
```

# PWM – Ejemplo cambio intensidad de LED

```
import RPi.GPIO as GPIO
from time import sleep

GPIO.setmode(GPIO.BCM)
GPIO.setup(18, GPIO.OUT, initial=GPIO.LOW) # LED GPIO 18

pin_pwm = GPIO.PWM(18, 200) #configurado a 200Hz, por ejemplo

pwm_value = 0 #valor inicial. El valor oscila entre 0 y 100
pin_pwm.start(pwm_value)

pin_pwm.ChangeDutyCycle(50) #cambiamos la intensidad a la mitad
sleep(1)

pin_pwm.ChangeDutyCycle(100) #cambiamos la intensidad al máximo
sleep(1)

pin_pwm.stop() #Paramos el pulso PWM
GPIO.cleanup() #Limpiamos los pines
```

# Eventos con GPIOs

- ¿Para qué los necesitamos?

Imaginad que queremos detectar que un pulsador se ha activado

- A. continuamente, en un bucle, comprobar si el pin donde está conectado tiene valor alto

# Eventos con GPIOs

- Polling: detectar pulsador activado “continuamente”

```
import RPi.GPIO as GPIO
from time import sleep # Import the sleep function from the time module

GPIO.setmode(GPIO.BCM)
GPIO.setup(19, GPIO.IN) # Pulsador GPIO-19

while True:
    inputValue = GPIO.input(19)
    if (inputValue == True):
        print("Button press ")
        sleep(0.2)
```

¿qué problemas puede tener esta implementación?

# Eventos con GPIOs

- ¿Para qué los necesitamos?

Imaginad que queremos detectar que un pulsador se ha activado

- A. continuamente, en un bucle, comprobar si el pin donde está conectado tiene valor alto
- B. Activando una función “únicamente” cuando se active el pulsador**

# CALLBACKS – The Hollywood Principle





# CALLBACKS

- El concepto de un callback es informar a una clase de que el trabajo de otra clase se ha terminado
- En vez de hacer un “poll” y continuamente comprobar si se ha terminado el trabajo, con un callback, se llama “automáticamente” a una función cuando se ha terminado el trabajo

# Callbacks con GPIOs

- Con la librería en Python

```
def procesar(numcanal):  
    print("Detectado")
```

```
GPIO.add_event_detect(19, GPIO.RISING, callback=procesar)
```

**IMPORTANTE:** se ejecutan en una hebra diferente a la del main(), así que si se quiere compartir variables Entre ambas threads, hay que asignarlas como globales.

- **Importante también definir la función antes de asociarla al evento!!**

# Variables compartidas entre threads: ejemplo

```
myVariableCompartida = 0

def procesar(numcanal):
    global myVariableCompartida
    myVariableCompartida = 1
    print("Detectado")

GPIO.add_event_detect(19, GPIO.RISING, callback=procesar)

while True:
    global myVariableCompartida
    if myVariableCompartida == 1:
        print(myVariableCompartida)
```

# Callbacks con GPIOs

- Con la librería en Python

```
GPIO.add_event_detect(19, GPIO.RISING, callback=procesar)
```

```
def procesar(numcanal):  
    print("Detectado")
```

GPIO.FALLING  
GPIO.BOTH

# Callbacks con GPIOs

- Múltiples *callbacks*

```
GPIO.add_event_detect(channel, GPIO.RISING, my_callback_one)  
GPIO.add_event_detect(channel, GPIO.FALLING, my_callback_two)
```

- Las funciones de callback se ejecutan de manera secuencial, no concurrente
- Esto ocurre porque hay una única hebra reservada para los callbacks.
- Los callbacks se ejecutarán en el orden en que están declarados

# wait\_for\_edge() function

BLOQUEA la ejecución del programa hasta que se detecte un flanco

```
# Espera hasta 5 segundos a un flanco de subida (timeout está en milisegundos)
channel = GPIO.wait_for_edge(channel, GPIO_RISING, timeout=5000)
if channel is None:
    print('Timeout occurred')
else:
    print('Edge detected on channel', channel)
```

# Event\_detected() function

Diseñada para usarse dentro de un bucle con otras tareas.  
A diferencia del callback, la actuación/respuesta al flanco no es inmediata

```
GPIO.add_event_detect(channel, GPIO.RISING) # add rising edge detection on a channel

do_something()

if GPIO.event_detected(channel):
    print('Button pressed')
```

# Bouncing effect

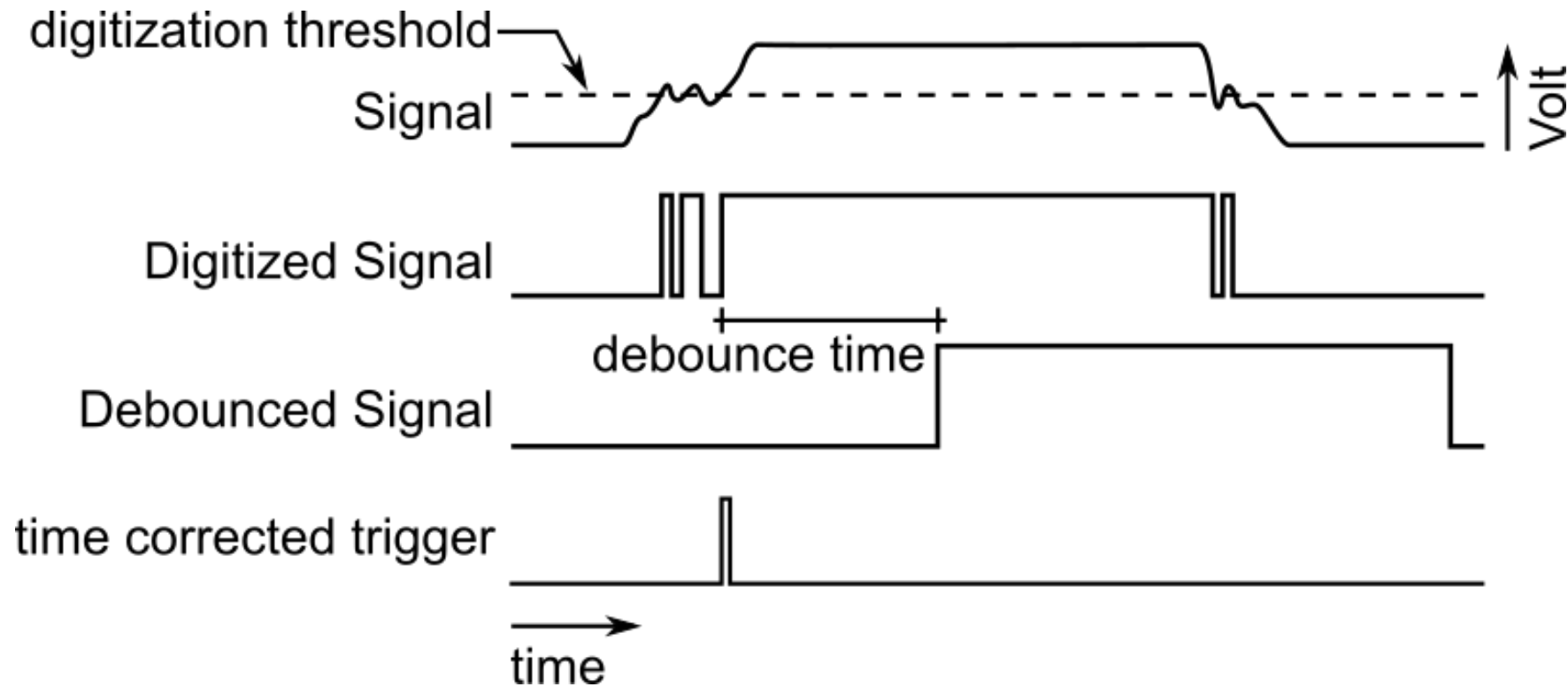
## Problema: *Bouncing effect*

- Los pulsadores a menudo generan transiciones open/close espúereas, debido a componentes mecánicos y físicos. Estas **transiciones se leen como varias pulsaciones en un tiempo muy corto**, ocasionando un mal funcionamiento del programa.
- Solución: comprobar más de una vez en un tiempo corto que se ha pulsado el botón, y considerarla como una única pulsación/activación



# Bouncing effect

## Problema: Bouncing



# Bouncing effect

## Problema: Bouncing

- **Bouncing**

Tendencia de 2 contactos metálicos en un dispositivo electrónico a generar múltiples señales cuando estos contactos se abren o cierran

- **debouncing**

Cualquier dispositivo software o hardware que se asegura que únicamente 1 señal se active para un único par de contactos abierto o cerrado

# Bouncing effect

- Ejemplo de callback, con *debounce*

```
import RPi.GPIO as GPIO
from time import sleep

GPIO.setmode(GPIO.BCM)
GPIO.setup(19, GPIO.IN, pull_up_down=GPIO.PUD_UP) # Pulsador GPIO-19

GPIO.add_event_detect(19, GPIO.RISING, callback=procesar, bouncetime=200)

def procesar(numcanal):
    print("Detectado")

while True:
    sleep(0.1)
```

# Callbacks con GPIOs

- Ejemplo de callback, con **debounce**. Encender LED al pulsar el pulsador

```
import RPi.GPIO as GPIO
from time import sleep # Import the sleep function from the time module

GPIO.setmode(GPIO.BCM)
GPIO.setup(14, GPIO.IN, pull_up_down=GPIO.PUD_UP) # Pulsador GPIO-14
GPIO.setup(18, GPIO.OUT, initial=GPIO.HIGH) # LED GPIO 18

GPIO.output(18, GPIO.HIGH) # Turn off. Valor inicial

GPIO.add_event_detect(14, GPIO.RISING, callback=procesar, bouncetime=200)

def procesar(numcanal):
    print("Detectado")
    GPIO.output(18, GPIO.LOW) # Turn on

while True:
    sleep(0.2)
    GPIO.output(18, GPIO.HIGH) # Turn off
```

# Callbacks con GPIOs

- Para cancelar el callback u otros eventos asociados a un GPIO:

```
GPIO.remove_event_detect(numero_del_pin)
```

# ACTIVIDAD DE CLASE

## SENSOR DE TEMPERATURA Y HUMEDAD

