introduccion.py clase08.py

```
Programming ‘Language’ = {
    Introducción = [Python, MicroPython]

}
```

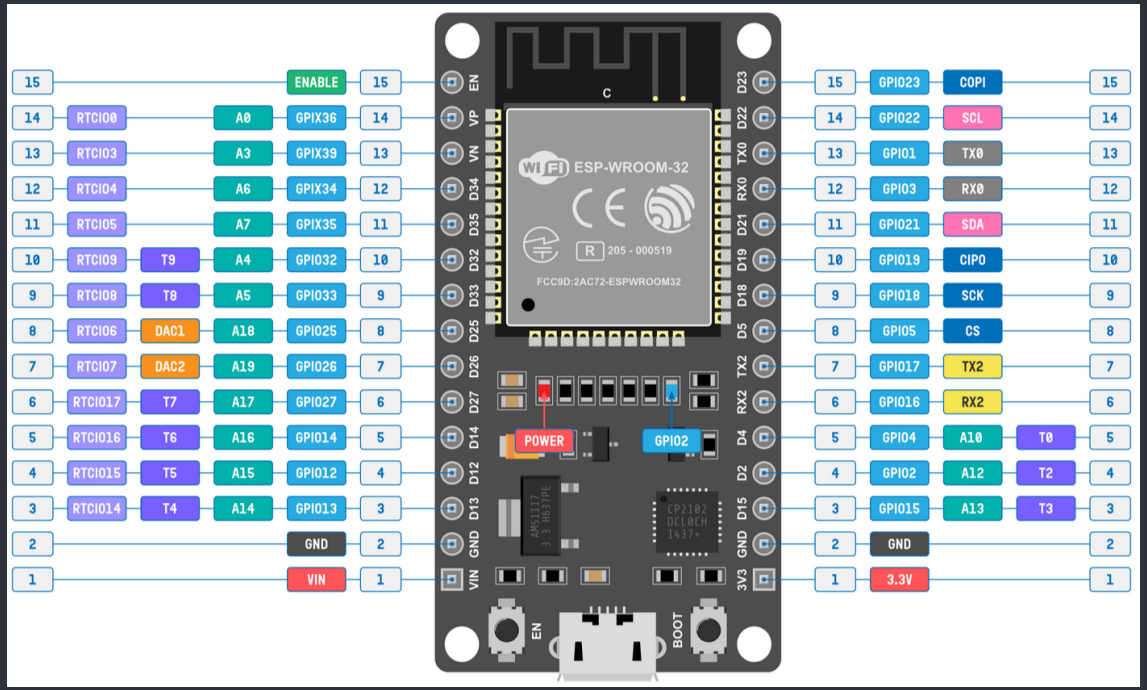
introduccion.py clase08.py

```
Clase 08 = {
    Presentación = [Les damos la bienvenida al curso]
}
```

Clases = {

- **Clase 05** — Conceptos Básicos de POO. Clases y objetos. Métodos y atributos.
- **Clase 06** — Instalación de MicroPython en la placa ESP32. Introducción a la herramientas de desarrollo Thonny. Conexión y configuración de la placa.
- **Clase 07** — Control de Hardware Básico. Manejo de pines GPIO. Lectura de sensores y actuadores.
- 👉 **Clase 08** — **Comunicación Serial. UART, I²C, SPI. Comunicación entre dispositivos.**

ESP32 DevKit V1 Pinout {



}

Buses y topologías {

Los circuitos electrónicos suelen comunicarse entre ellos usando buses de comunicaciones. Esto quiere decir, por ejemplo, que para que dos microcontroladores interactúen entre ellos o con sus periféricos, se suelen utilizar buses. En electrónica entender estos conceptos es **fundamental** para poder realizar circuitos digitales funcionales.

Un bus de comunicaciones es un canal físico de transmisión que permite la comunicación entre distintos dispositivos o componentes de un sistema, como un microcontrolador y sus periféricos.

Un bus de comunicación a su vez puede dividirse en buses de **datos**, de **direcciones**, de **control**, etc. Estos buses nos sirven, por ejemplo, para transferir datos, sincronizar señales, controlar dispositivos. Dependiendo del bus, será el protocolo que se usará.

Existen varias topologías, pero las dos mas ampliamente usadas son la topología **Serie** y la topología **Paralelo**.

}

Buses y topologías {

```
# En la topología Serie los datos se transfieren bit a bit a través de un solo canal, mientras que en la topología Paralelo los datos se transfieren varios bits a la vez a través de múltiples canales.

# En ambos casos, vamos a poder identificar en toda comunicación las siguientes partes fundamentales.

# Emisor: es quien emite el mensaje o el dato que se quiere transmitir.

# Receptor: es quien recibe el mensaje.

# Mensaje: es el dato que se envía y tanto el receptor como el emisor deben hablar el mismo “idioma” o “protocolo” para poder interpretarlo.

# Canal (capa física): Es el medio por el cual se envía el mensaje, puede ser un cable, una pista en una placa, el aire, etc.
```

}

Topología Paralelo {

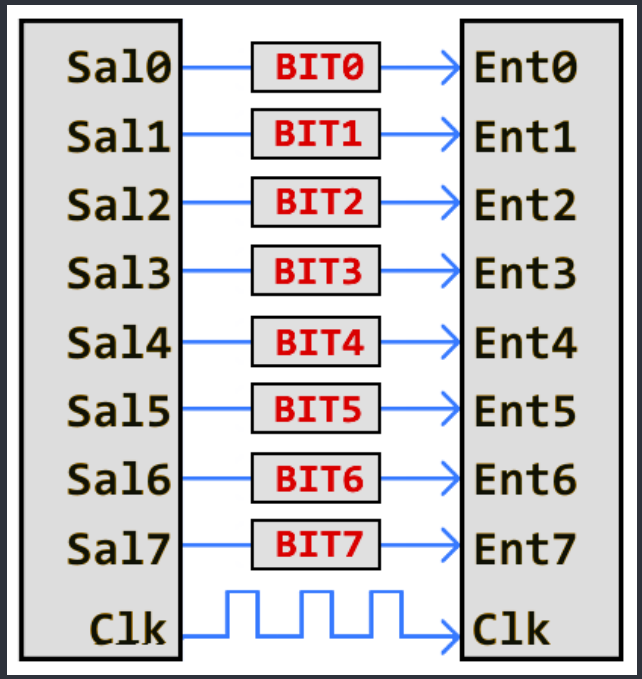
Utiliza múltiples canal para transmitir datos simultáneamente. **Mayor velocidad de transferencia de datos.**

Distancia de comunicación corta debido a la sincronización de señales. **Menor latencia en transferencia de datos.**

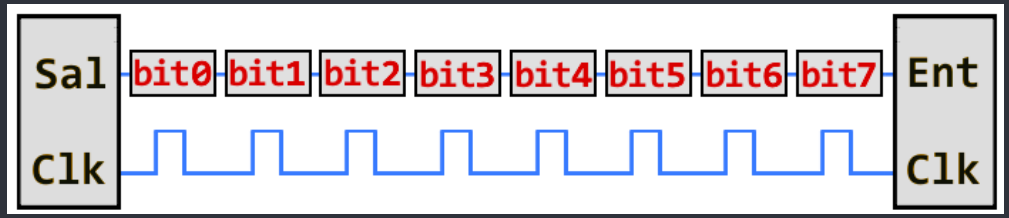
Mas costoso por complejidad y cantidad de líneas necesarias. **Mayor interferencia** electromagnética. **No apto para distancias largas.**

Ejemplos: Buses de datos en computadoras, interfaces de impresoras antiguas, etc. (ISA, PCI, AGP, IDE, LPT, Centronics, LCDs paralelos, DVP, GPIOs).

}



Topología Serie {



Utiliza un solo canal y los datos se envían un bit a la vez. **Menor velocidad de transferencia de datos.**

Permite distancias de comunicación mayores. **Mayor latencia en transferencia de datos.**

Menos costoso por simpleza y cantidad de líneas necesarias. **Menor interferencia** electromagnética.

}

Ejemplos: UART, I^2C, SPI, WiFi, USB, SATA, Bluetooth, CAN, Modbus.

Comparativa {

Característica	Serial	Paralelo
Transmisión de datos	Secuencial (bit por bit)	Simultánea (varios bits a la vez)
Número de líneas	2-4 (generalmente)	Múltiples (8, 16, o más)
Distancia de transmisión	Larga	Corta
Costo y complejidad de cableado	Bajo y simple	Alto y complejo
Velocidad de transmisión	Menor en general, pero eficiente para largas distancias	Alta en distancias cortas
Interferencia	Baja	Alta
Sincronización	Simple	Compleja
Casos de uso	Comunicación entre dispositivos, sensores, IoT, inalámbrica, etc	Buses internos de la PC, datos de alta velocidad como video, etc.

```
# Los buses paralelos sentaron las bases para muchos de los estándares
actuales en conectividad y fueron cruciales en el desarrollo de la
tecnología de comunicaciones en hardware. Debido a las limitaciones de
espacio en la mayoría de los microcontroladores y la eficiencia de los
buses seriales como SPI e I²C, los protocolos paralelos han sido en gran
medida reemplazados.
```

}

UART {

UART (**universal asynchronous receiver / transmitter**) define un protocolo o un conjunto de normas para el intercambio de datos en serie entre dos dispositivos.

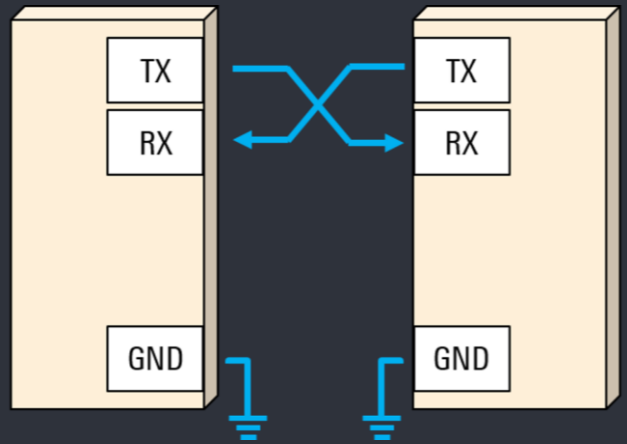
UART es sumamente simple y utiliza solo **dos hilos** entre el transmisor y el receptor para transmitir y recibir en ambas direcciones. Ambos extremos tienen una conexión a masa. La comunicación en UART puede ser **simplex** (los datos se envían en una sola dirección), **semidúplex** (cada extremo se comunica, pero solo uno al mismo tiempo), o **dúplex completo** (ambos extremos pueden transmitir simultáneamente). En UART, los datos se transmiten en forma de tramas.

UART fue uno de los primeros protocolos en serie. Los puertos en serie, que en su día proliferaron a gran escala, se basan casi siempre en el protocolo UART, y los dispositivos que utilizan interfaces RS-232, módems externos, etc. son ejemplos típicos de la aplicación de UART.

}

UART {

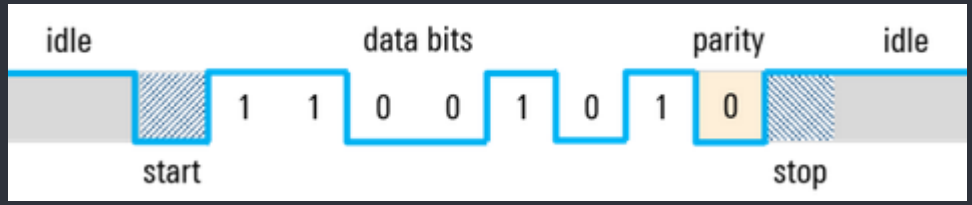
```
# Una de las mayores ventajas de UART es que es asíncrono: el transmisor y el receptor no comparten la misma señal de reloj. Si bien esto simplifica en gran medida el protocolo, plantea determinados requisitos en el transmisor y el receptor. Puesto que no comparten un reloj, ambos extremos deben transmitir a la misma velocidad, previamente concertada, con el fin de mantener la misma temporización de los bits.
```




```
# Las velocidades en baudios más habituales en UART que se utilizan actualmente son 4800, 9600, 19,2 K, 57,6 K, y 115,2 K. Además de tener la misma velocidad en baudios, ambos extremos de una conexión UART deben utilizar también la misma estructura y parámetros de trama.
```

}

UART {




```
    # Ejemplo de trama y sus parámetros, que deben ser predefinidos entre
    emisor y receptor previamente para su correcta interpretación. Vemos:

    # Bit de inicio o start (al ser asíncrono es necesario especificar cuando
    comienza la transmisión de los datos).

    # Bits de datos (normalmente entre 5 y 9 bits)

    # Bit de paridad (opcional, para verificar errores, puede ser par o impar)
}
    # Bit de fin o stop (puede ser 1 o 2 bits)
```

SPI {

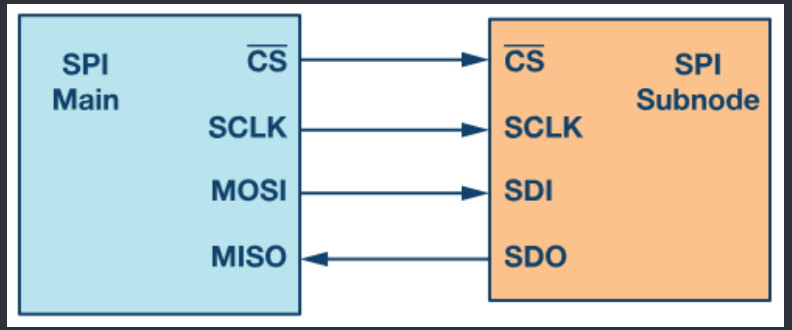
SPI es un protocolo de comunicación serial síncrono que utiliza al menos cuatro cables: **MOSI** (Master Out Slave In), **MISO** (Master In Slave Out), **SCK** (reloj) y **CS/SS** (Chip Select/Slave Select). Permite la comunicación rápida entre un maestro y múltiples esclavos.

SPI fue desarrollada por Motorola (ahora NXP Semiconductors) aproximadamente en 1979. Es una interfaz síncrona de comunicación rápida a corta distancia.

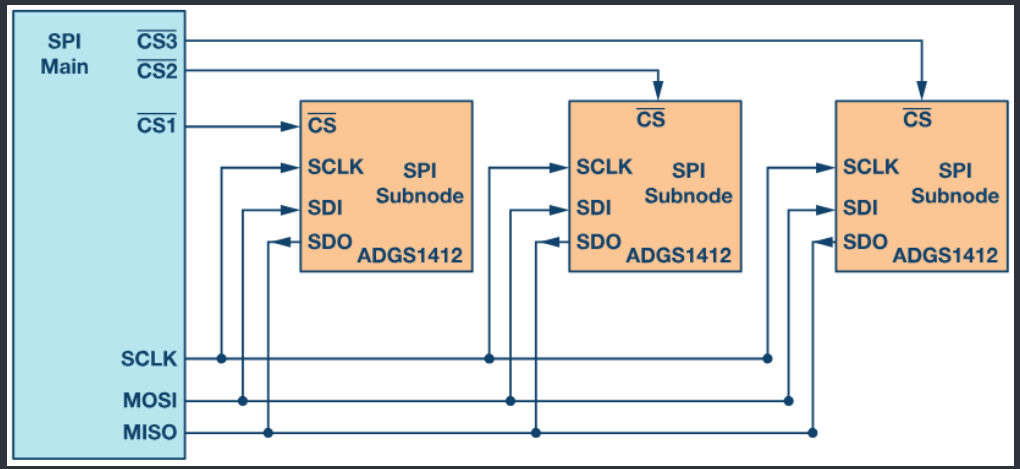
Usa un modelo de maestro-esclavo con un maestro simple y puede manejar varios dispositivos esclavos usando comunicaciones full dúplex que operan a velocidades de reloj de hasta 50 MHz. No usa un protocolo estándar y transfiere solo paquetes de datos, lo que la hace ideal para transferir flujos de datos largos.

}

```
SPI {

    # Las señales son:

    # MOSI/SDI (Master Out Slave In)

    # MISO/SDO (Master In Slave Out)

    # SCK/SCLK (reloj)

    # CS/SS (Chip Select/Slave
    Select)

    # El protocolo SPI no cuenta con un mecanismo de confirmación de datos
    como bits de paridad en el caso de UART o el ACK/NACK de I²C. Debido a
    esto, en SPI, la detección de problemas de comunicación debe hacerse de
    otras maneras, ya que el protocolo en sí no garantiza que el receptor haya
    recibido los datos correctamente.
}
```



SPI {

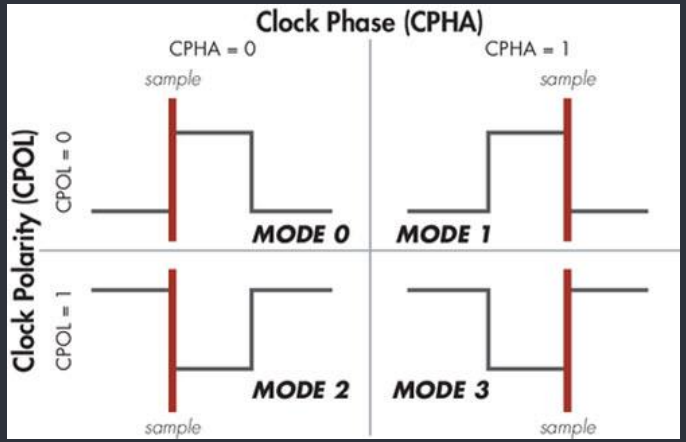



```
# Se pueden conectar tantos dispositivos esclavos como se desee, teniendo
en cuenta que se va a necesitar, además de los pines MOSI, MISO y SCK, un
pin en el dispositivo maestro por cada dispositivo esclavo que se conecte.
```

}

SPI {

Modo SPI	CPOL	CPHA	Polaridad de clock (valor en estado inactivo)	Fase del reloj para toma de dato y entrega de dato
0	0	0	0	Tomado en **flanco ascendente** y colocado en el **flanco descendente**.
1	0	1	0	Tomado en flanco **descendente** y colocado en el flanco **ascendente**.
2	1	0	1	Tomado en flanco **descendente** y colocado en el flanco **ascendente**.
3	1	1	1	Tomado en **flanco ascendente** y colocado en el **flanco descendente**.




```
# En SPI solo se configura el modo, el cual puede ser 0, 1, 2 o 3.
```

}

I^2C {

I^2C es un protocolo de comunicación serial síncrono que utiliza dos cables: **SDA/SDL** (Serial Data Line) y **SCL/SCK** (Serial Clock Line).

Fue desarrollado por Philips Semiconductor (ahora NXP Semiconductors) en la década de 1980 para simplificar la comunicación entre los microcontroladores y los periféricos en un sistema.

Un dispositivo maestro controla el reloj general y coordina la comunicación entre varios esclavos. A diferencia de SPI, que tenemos que habilitar el dispositivo con el cual queremos interactuar, en I^2C todos los dispositivos están activos, y vamos a interactuar con ellos según su dirección asignada.

Ambas líneas SDA y SCL necesitan resistencias pull-up externas para mantener la señal en estado alto cuando no se usa (esto es porque I^2C usa nivel bajo para transmitir bits). Cuando se comunica, el dispositivo activa la línea llevándola a bajo para indicar un bit de datos.

}

I²C {

Procedimiento:

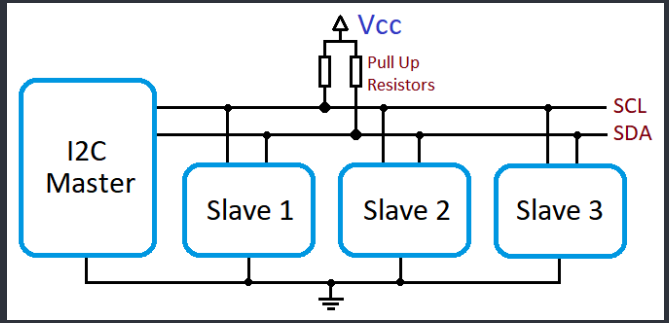
1 - El maestro genera el reloj en la línea SCL.

2 - El maestro inicia la comunicación enviando una condición de inicio (Start Condition).

3 - Luego envía una dirección de 7 o 10 bits para seleccionar un esclavo.

4 - Los datos se transmiten en paquetes de 8 bits (bytes), con una señal de ACK (acknowledge) o NACK (no acknowledge) después de cada byte para confirmar la recepción correcta.

}



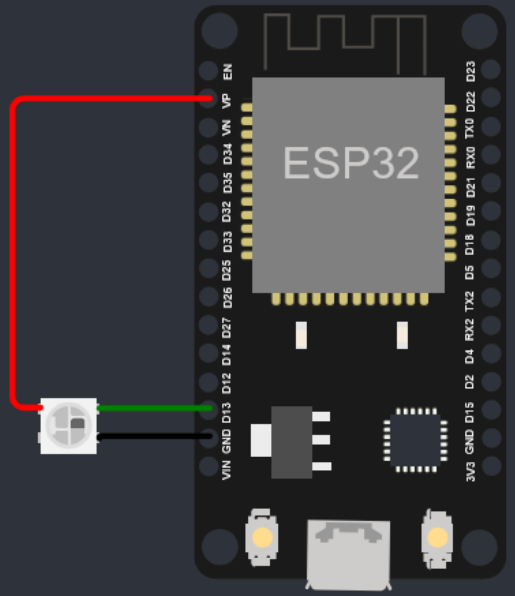
UART vs SPI vs I²C {

```
# UART: Principalmente usado para comunicación directa entre dos dispositivos, ideal para transmisiones de datos simples como configuración de periféricos o comunicación en microcontroladores. Se sigue usando pero no es común verlo en dispositivos modernos. JTAG es una alternativa que se ve mucho mas hoy en día.

# I²C: Adecuado para sensores y periféricos de baja velocidad, se usa mucho en sistemas integrados con varios sensores y dispositivos de baja velocidad.

# SPI: Ideal para aplicaciones que requieren una transmisión de datos rápida y en tiempo real, como pantallas y almacenamiento rápido.
```

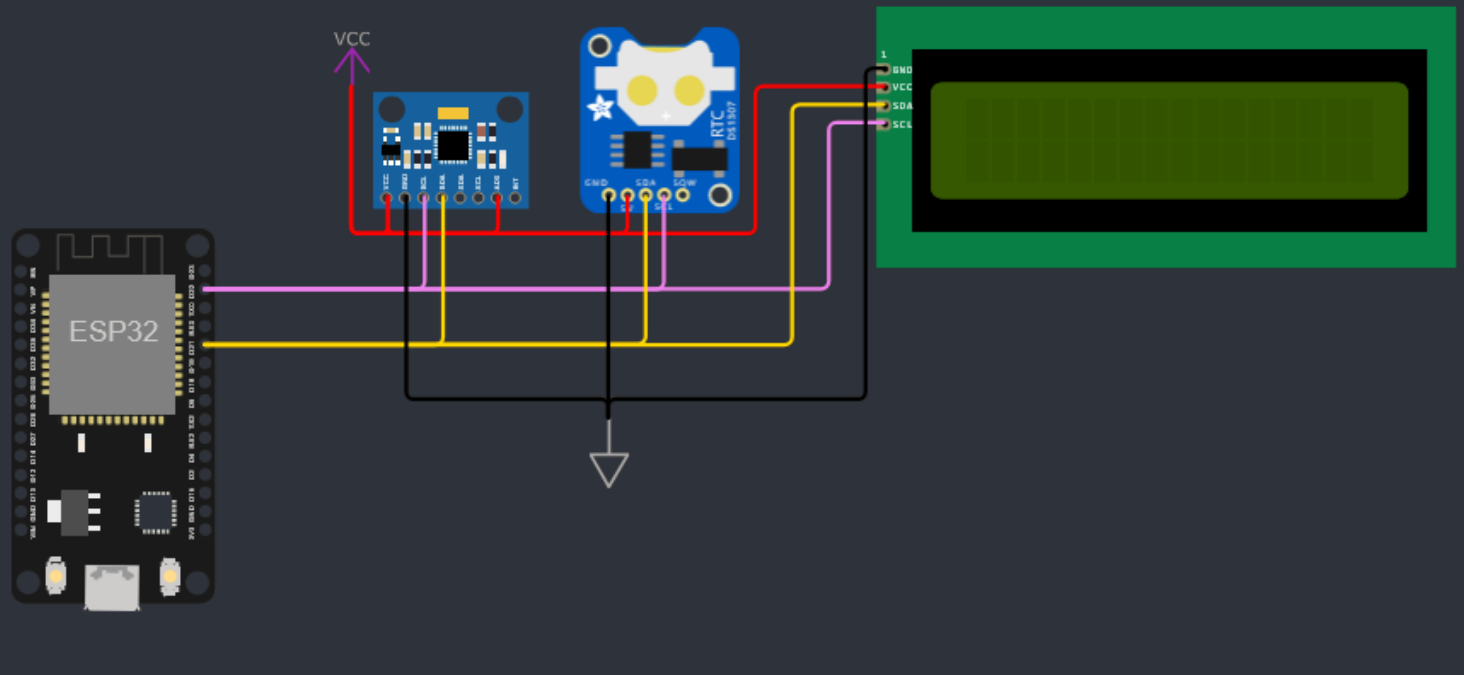
}

Ejemplo 1 – Neopixel {

```
1
2
3
4   import machine
5   from neopixel import NeoPixel
6
7   #Crea un objeto led de la clase NeoPixel en
    el pin 16
8   led = NeoPixel(machine.Pin (13),1)
9
10  led[0]=(255, 0, 0)
11  led.write ()
12
13
14 }
```



Ejemplo 6 – Escaneo de I²C {



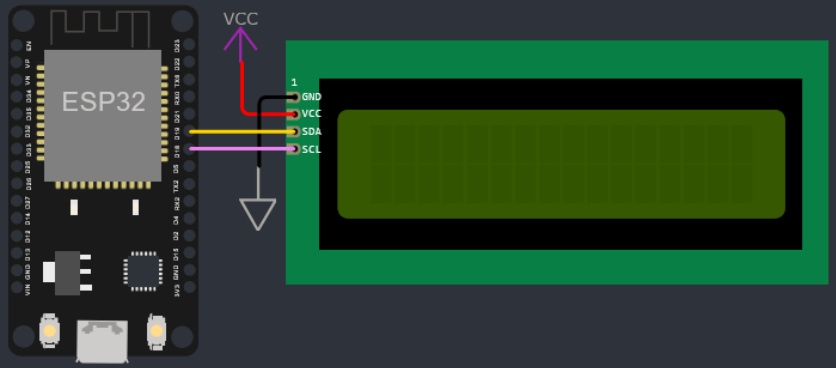
}

Ejemplo 6 – Escaneo de I²C {

```
import machine
i2c = machine.I2C(0, scl=machine.Pin(22), sda=machine.Pin(21))

print('\033[2J\033[H') # Limpiamos la pantalla
print('Escaneando bus I2C en busca de dispositivos...\n')
devices = i2c.scan()

if len(devices) == 0:
  print("No se encontraron dispositivos I2C conectados.\n")
else:
  print(f'Se encontraron {len(devices)} dispositivos conectados.\n')

  for i in range(len(devices)):
    print(f'Dispositivo {i+1}, direccion en decimal: {devices[i]} |
direccion en hexa {hex(devices[i])}')
```

}

Ejemplo 7 – LCD 2x16{

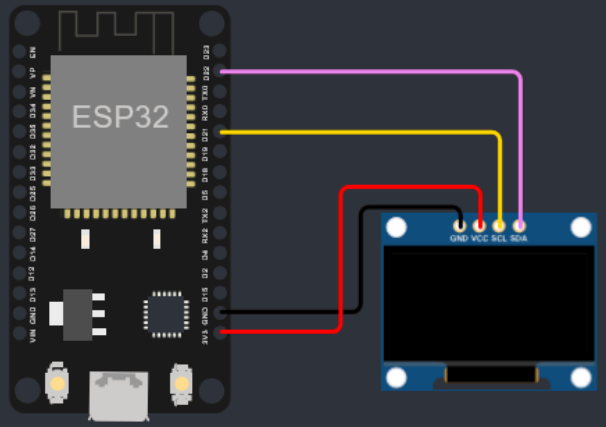



```python
import machine
from lcd_api import LcdApi
from i2c_lcd import I2cLcd
i2c = machine.SoftI2C(sda=machine.Pin(19), scl=machine.Pin(18), freq=1000000)
lcd = I2cLcd(i2c, 0x27, 2, 16)
lcd.move_to(2,0)
lcd.putstr("Micropython!")
lcd.move_to(4,1)
lcd.putstr("UTN-FRT!")
}
```

Ejemplo 11 – OLED SSD1306 {

```
from machine import Pin, SoftI2C
from ssd1306 import SSD1306_I2C

i2c = SoftI2C(scl=Pin(21), sda=Pin(22),
freq=1000000)

oled_width = 128
oled_height = 64
oled = SSD1306_I2C(oled_width, oled_height,
i2c)

oled.text('Hola curso!', 10, 5)
oled.text('Micropython', 20, 25)
oled.text('UTN - FRT', 30, 45)
oled.show()
```

}



1
2
3 Aprender a programar
4
5 es aprender a pensar.
6
7
8
9
10
11 { Steve Jobs; }
12
13
14

{ Nos vemos en la proxima clase }