comunicación entre Arduino Uno y Raspberry Pi Pico

Mateo Gutiérrez (Cód. 2334475)
Julián Montealegre (Cód. 2343485)
Nicolás Alfonso (Cód. 2303507)

***Resumen*—En este laboratorio se desarrolló la comunicación entre una placa Arduino y una Raspberry Pi, con el objetivo de establecer un intercambio de información confiable entre ambos dispositivos. Para ello, se configuraron los puertos de comunicación y se implementaron protocolos adecuados para el envío y recepción de datos, asegurando la correcta interpretación de la información transmitida. Esta práctica permitió comprender las diferencias arquitectónicas entre un microcontrolador y un sistema en un chip (SoC), así como las ventajas y limitaciones de cada uno al trabajar de forma conjunta en un sistema embebido.**

***Index Terms*—Arduino, Raspberry Pi, Microcontrolador, SoC, Arquitectura de computadoras, UART, SPI, I²C**

I. Marco teórico

I-A. Arquitectura del Arduino Uno

I-A1. Componentes principales del ATmega328P: El microcontrolador ATmega328P utilizado en Arduino Uno integra los siguientes componentes clave:

- **CPU**: Unidad de procesamiento de 8-bit AVR
- **Memoria**: 32KB Flash, 2KB SRAM, 1KB EEPROM
- **Periféricos**:
  - 2 temporizadores/contadores de 8-bit
  - 1 temporizador/contador de 16-bit
  - 6 canales PWM
  - Convertidor ADC de 10-bit (6 canales)
- **Comunicaciones**: UART, SPI, I²C
- **GPIO**: 23 líneas de E/S digitales

Estos componentes interactúan mediante un bus de datos de 8-bit y bus de direcciones de 16-bit, coordinados por la unidad de control de la CPU [1].

I-A2. Manejo de interrupciones: El sistema de interrupciones en Arduino se caracteriza por:

- Arquitectura de vectores de interrupción
- 26 fuentes de interrupción (2 externas)
- Latencia típica de 4 ciclos de reloj
- Prioridad fija por posición en vector

La secuencia típica incluye:

1. Finalización de instrucción en curso
2. Guardado de contexto (PC → pila)
3. Ejecución de ISR (Rutina de Servicio)
4. Restauración de contexto (RETI)

I-A3. Reloj de 16 MHz:

- Frecuencia óptima para balancear:

$$P_{dyn} = C_{eff} \cdot V_{dd}^2 \cdot f_{clk} \tag{1}$$

- Velocidad de instrucciones típicas: 1 ciclo (ALU) a 4 ciclos (acceso memoria)
- Throughput máximo: ∼16 MIPS

I-B. Arquitectura de Raspberry Pi

I-B1. Comparación con microcontroladores: Las diferencias fundamentales entre Raspberry Pi (SoC Broadcom BCM2835/7) y Arduino se resumen en la Tabla I:

Cuadro I
Comparación Arduino vs Raspberry Pi

Característica	Arduino Uno	Raspberry Pi
Arquitectura	Harvard (8-bit)	Von Neumann (32/64-bit)
Frecuencia	16 MHz	1.2–1.5 GHz
Memoria	2KB SRAM	1–8GB SDRAM
GPIO	Digital + Analog	Digital (requiere ADC)
SO	No	Linux-based

I-B2. Sistema Operativo y GPIO: La necesidad de SO en Raspberry Pi deriva de:

- Gestión de memoria virtual (MMU)
- Soporte para multitarea preventiva
- Drivers para periféricos complejos

El subsistema GPIO ofrece:

- 40 pines (26 en modelos tempranos)
- Funciones alternativas (UART, SPI, I2C)
- Control por memoria mapeada (registros)

I-B3. ADC: Arduino vs Raspberry Pi:

- **Arduino**:
  - ADC integrado de 10-bit (0–1023)
  - Tasa de muestreo ∼10 kSPS
  - Precisión ±2 LSB
- **Raspberry Pi**:
  - Requiere ADC externo (ej. MCP3008)
  - Comunicación vía SPI/I2C
  - Resolución típica 12–16 bits

II. Metodología y procedimientos

II-A. Punto 2: UART con detección de errores

Se implementó la comunicación serie asíncrona (UART) entre un Arduino Uno y una Raspberry Pi, utilizando paridad como mecanismo de detección de errores.

- **Arduino TX → Raspberry Pi RX (GPIO 15)**
- **Arduino RX → Raspberry Pi TX (GPIO 14)**

- **GND Arduino ↔ GND Raspberry Pi**

Indicadores visuales en la protoboard: LED verde (dato válido) y LED blanco (error de paridad). El código fuente se encuentra en **Anexos/GitHub**.

Procedimiento detallado:

1. Configuración del puerto serie en ambos dispositivos (formato 8 bits de datos, 1 bit de parada y paridad habilitada).
2. Envío periódico de caracteres desde Arduino; recepción en Raspberry Pi con verificación de paridad.
3. Mapeo del resultado de la verificación a LEDs (verde = válido, blanco = error).
4. Pruebas de robustez: variación de distancia/cables y observación de eventos de error en consola y LEDs.

Criterios de validación:

- Mensajes en consola coherentes con "dato válido"/"error de paridad".
- Correspondencia 1:1 entre los eventos reportados y el estado de los LEDs.

Riesgos y mitigaciones:

- Desacople de GND o ruidos: cableado corto y buena referencia de tierra.
- Configuración de *baudrate* no coincidente: verificación previa con bucle local de eco.

II-B. Punto 3: SPI sincrónico con control de LED

Se utilizó SPI con la Raspberry Pi como maestro y el Arduino Uno como esclavo. Conexiones estándar: MOSI, MISO, SCK y SS/CE0, además de GND común. El LED del Arduino (pin 9 con resistencia de 220Ω) fue controlado por los comandos:

- `0x01`: encender LED
- `0x00`: apagar LED

La Pi envió comandos periódicamente y recibió confirmación por MISO. El código se documenta en **Anexos/GitHub**.

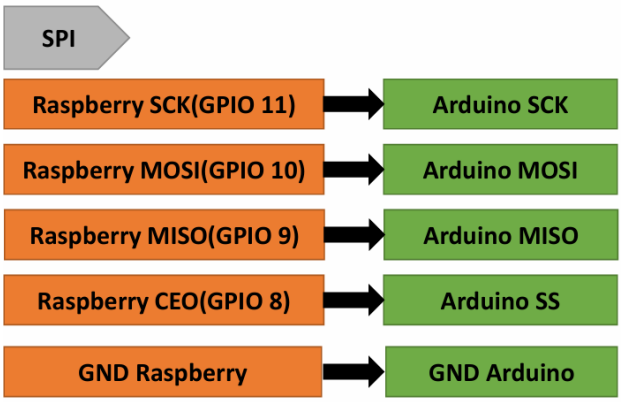

Figura 1. Diagrama de conexiones SPI entre Raspberry Pi y Arduino.

Procedimiento detallado:

1. Activación del bus SPI en Raspberry Pi y prueba de dispositivo.
2. Configuración del esclavo SPI en Arduino; habilitación de la interrupción de transferencia.
3. Envío alternado de comandos de control; lectura de la respuesta del esclavo.
4. Verificación temporal: el LED cambia de estado dentro del periodo programado.

Criterios de validación:

- Respuestas del esclavo coherentes con el comando enviado.
- Cambios de LED reproducibles y observables en fotos y consola.

Riesgos y mitigaciones:

- Conexiones cruzadas (MOSI/MISO): rotulado de cables y prueba de eco.
- Frecuencia de reloj excesiva: uso de valores conservadores y aumento gradual.

II-C. Punto 4: I^2C con potenciómetro y LEDs

Se estableció comunicación I^2C con Arduino como esclavo y Raspberry Pi como maestro. Conexiones:

- **SDA**: GPIO 2 (Raspberry Pi) ↔ A4 (Arduino)
- **SCL**: GPIO 3 (Raspberry Pi) ↔ A5 (Arduino)
- **GND**: referencia común

El potenciómetro se conectó a 5V y GND; terminal central a A0. La Raspberry Pi mostró los **tres bits más significativos** en los GPIO 22, 23 y 24 mediante LEDs con resistencias de 220Ω. El código se incluye en **Anexos/GitHub**.

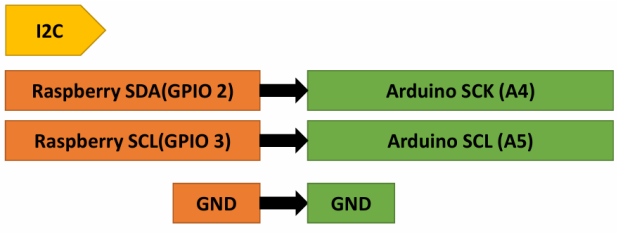

Figura 2. Diagrama de conexión I^2C con potenciómetro y LEDs.

Procedimiento detallado:

1. Configuración de Arduino como esclavo I^2C (dirección `0x08`); lectura de A0 con ADC de 10 bits.
2. Solicitud periódica del valor por parte de la Raspberry Pi (maestro).
3. Extracción de los 3 MSB y mapeo a los GPIO (22, 23, 24) para LEDs.
4. Impresión en consola del valor completo para verificación.

Criterios de validación:

- Cambio coherente de LEDs al girar el potenciómetro.
- Valor numérico en consola aumentando/disminuyendo de forma estable.

Riesgos y mitigaciones:

- Diferencia de niveles lógicos (5V ↔ 3.3V): uso de *pull-ups* a 3.3V y evitar *pull-ups* a 5V en el bus de la Raspberry Pi.
- Timeouts del maestro (p.ej., `Errno 110 ETIMEDOUT`): confirmar dirección `0x08`, revisar *pull-up* (4.7k–10k), y velocidad estándar.

III. Resultados

III-A. Resultados UART

Durante la práctica se logró establecer comunicación UART con detección de errores por paridad.

- **LED verde**: dato válido y paridad correcta.
- **LED blanco**: error de paridad en la transmisión.

```
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
Error de paridad! Byte: 0xc1
Dato válido: b'A'
```

Figura 3. Salida en terminal mostrando datos válidos y errores de paridad.

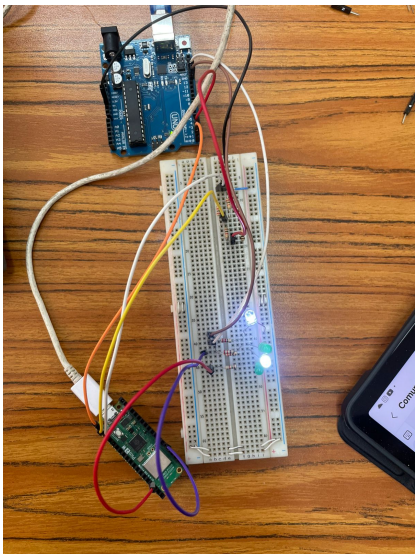

Figura 4. LED blanco encendido: error de paridad detectado.

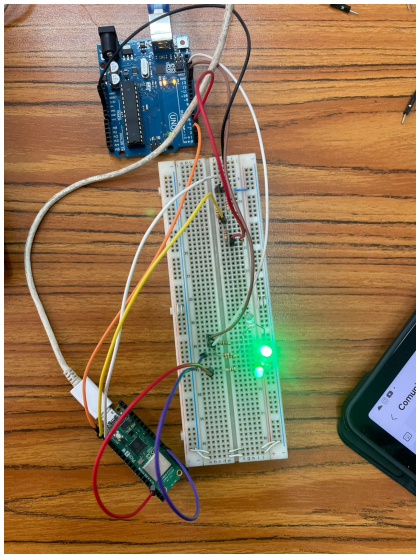

Figura 5. LED verde encendido: transmisión válida con paridad correcta.

III-B. Resultados SPI

Se comprobó el funcionamiento de la comunicación SPI entre la Raspberry Pi y el Arduino Uno.

- La Raspberry Pi envió comandos alternados `0x55` y `0xAA`, y comandos de control `0x00`/`0x01`.
- El Arduino encendió/apagó el LED según el comando recibido.
- La respuesta del esclavo se mostró en la consola de la Raspberry Pi.

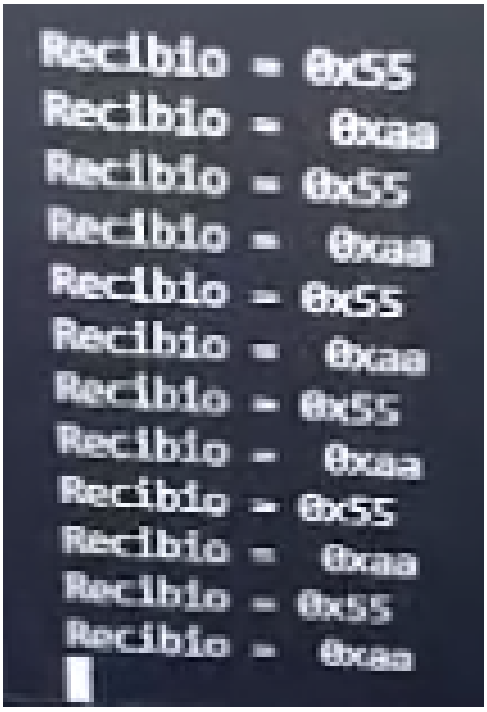

Figura 6. Consola de Raspberry Pi mostrando recepción alternada de datos vía SPI.

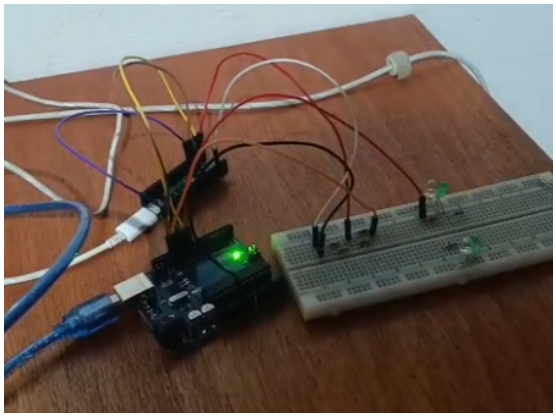

Figura 7. LED apagado: comando de apagado recibido vía SPI.

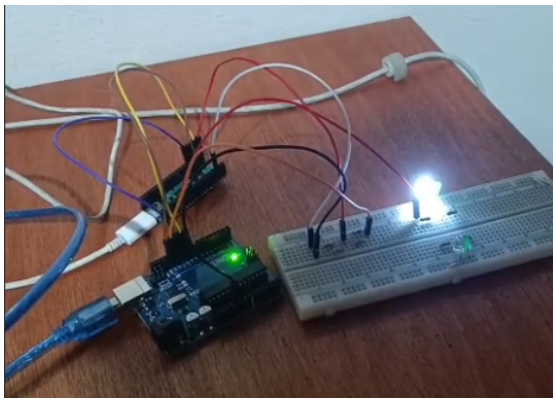

Figura 8. LED encendido: comando de encendido recibido vía SPI.

III-C. *Resultados I^2C*

Se validó la transmisión del valor del potenciómetro (A0) desde Arduino a Raspberry Pi y su visualización en LEDs.

- El valor varió dinámicamente al girar el potenciómetro.
- La Raspberry Pi actualizó los LEDs según los 3 MSB del valor recibido.

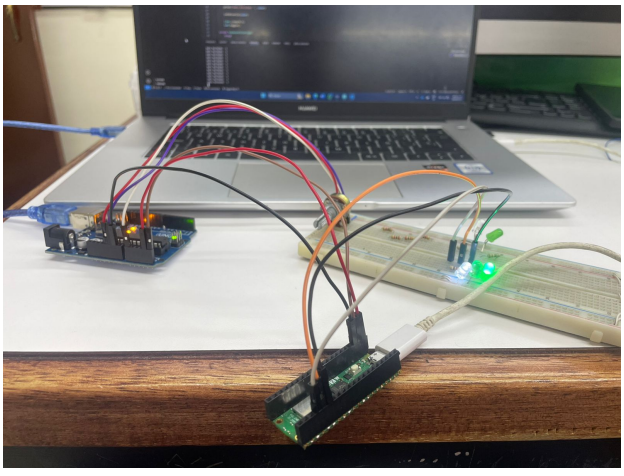

Figura 9. Valores altos: varios LEDs encendidos (MSB activos).

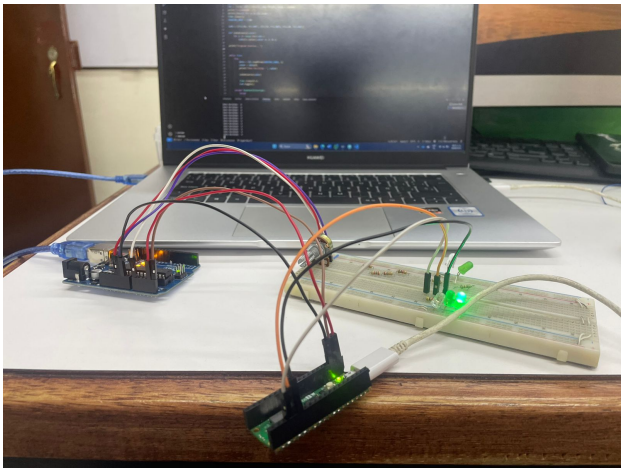

Figura 10. Valores bajos: menor número de LEDs encendidos.

IV. Observaciones y discusión

IV-A. *Protocolo UART*

- La paridad permitió detectar errores pero no corregirlos.
- La calidad de la referencia de tierra y el *baudrate* influyen en errores intermitentes.

IV-B. *Protocolo SPI*

- La señal de reloj compartida reduce desincronizaciones respecto a UART.
- Requiere más pines físicos (MOSI, MISO, SCK, SS), pero ofrece mayor velocidad y confiabilidad para control directo de periféricos.

IV-C. *Protocolo I^2C*

- Útil cuando varios dispositivos comparten el bus con solo dos líneas (SDA/SCL).
- Adecuado para sensado: envío de valores digitalizados (10 bits) y representación compacta (MSB a LEDs).
- Para integraciones con Raspberry Pi, asegurar *pull-ups* a 3.3V y revisar dirección de esclavo (`0x08`) ante timeouts.

V. Conclusiones

El análisis comparativo muestra que Arduino es óptimo para tareas deterministas de control y adquisición, mientras que Raspberry Pi aporta capacidad de procesamiento, sistema operativo y conectividad. Experimentalmente:

- **UART** facilitó una conexión punto a punto simple con detección básica de errores por paridad.
- **SPI** demostró ser veloz y robusto para control de periféricos, validado con el encendido/apagado del LED.
- **I^2C** evidenció eficiencia para sensado y buses compartidos, integrando lectura analógica y visualización digital.

El conjunto Arduino–Raspberry Pi constituye una plataforma híbrida efectiva para prácticas de IoT, domótica y robótica educativa.

REFERENCIAS

[1] ATMEL, “ATmega328P Datasheet,” 2016.
[2] Broadcom, “BCM2835 ARM Peripherals,” 2012.
[3] Arduino, “Arduino Uno Rev3 – Documentation,” Disponible en: https://docs.arduino.cc/
[4] Raspberry Pi Foundation, “Raspberry Pi Documentation,” Disponible en: https://www.raspberrypi.com/documentation/
[5] PySerial, “pySerial Documentation,” Disponible en: https://pyserial.readthedocs.io/
[6] Linux SPI, “spidev Userspace Interface,” Disponible en: https://www.kernel.org/doc/Documentation/spi/spidev
[7] Microchip, “MCP3008 10-Bit ADC Datasheet,” Disponible en: https://www.microchip.com/

APÉNDICE A
ANEXOS: CÓDIGO FUENTE

El código fuente utilizado en las prácticas se incluye a continuación. También está disponible en el repositorio de GitHub del proyecto.

A-A. Punto 1: UART con verificación de paridad

Raspberry Pi (MicroPython):

```
from machine import Pin, UART
import time

# Configura LEDs externos
led_verde = Pin(13, Pin.OUT)  # LED verde (dato válido)
led_rojo = Pin(21, Pin.OUT)   # LED rojo (error paridad)

# Configura UART (UART0: GP0=TX, GP1=RX)
uart = UART(0, baudrate=9600, bits=8, parity=1, stop=1)

try:
    while True:
        if uart.any():  # Si hay datos en el buffer
            data = uart.read(1)   # Leer 1 byte
            if data:
                byte_val = data[0]  # Convierte a entero (0-255)
                ones = bin(byte_val).count("1")
                is_valid = (ones % 2 == 0)  # Paridad par: nº de 1's debe ser par

                if is_valid:
                    led_verde.value(1)
                    print("Dato valido:", data)
                else:
                    led_rojo.value(1)
                    print("Error de paridad! Byte:", hex(byte_val))

                time.sleep(0.5)
                led_verde.value(0)
                led_rojo.value(0)

except KeyboardInterrupt:
    led_verde.value(0)
    led_rojo.value(0)
    print("Programa detenido")
```

Arduino:

```
void setup() {
  Serial.begin(9600, SERIAL_8E1);        // UART: 9600 baudios, 8E1
  pinMode(13, OUTPUT);       // LED verde (Arduino)
  pinMode(12, OUTPUT);       // LED rojo (Arduino)
}

void loop() {
  // Envío de byte válido: '0' = 0x30 = 00110000
  uint8_t Letra = 'A';
  Serial.write(Letra);

  digitalWrite(13, HIGH);   // LED verde ON (dato válido enviado)
  delay(500);
  digitalWrite(13, LOW);

  // Envío de byte con error simulado: 0xC1 = 11000001
  byte errorByte = 0xC1;
  Serial.write(errorByte);
  digitalWrite(12, HIGH);   // LED rojo ON (error simulado enviado)
  delay(500);
  digitalWrite(12, LOW);

  delay(1000);              // Pausa antes de repetir
}
```

A-B. Punto 2: SPI sincrónico con control de LED

Raspberry Pi (MicroPython):

```
from machine import SPI, Pin
import time

# SPI0:
# sck  (Clock)                  = GP18 => Pin 13
# mosi (Master Out - Slave in)  = GP19 => Pin 11
# miso (Master In - Slave Out)  = GP16 <= Voltage Divider <= Pin 12
# SS   (Slave Select)           = GP17 => Pin 10
# GND  (Ground)                 = GND <=> GND
# Led                           = Pin 9

spi = SPI(0, baudrate=1_000_000, polarity=0, phase=0,
          sck=Pin(18), mosi=Pin(19), miso=Pin(16))

cs = Pin(17, Pin.OUT)
cs.value(1)  # inactivo (alto)

commands = [0x01, 0x00]       # Encender y apagar LED

def Transfer(command):
    Buffer = bytearray(1)
    response = bytearray(1)
    Buffer[0] = command

    cs.value(0)
    spi.write_readinto(Buffer,response)
    cs.value(1)

    return response[0]

try:
    while True:
        response = Transfer(commands[0])
        print("Recibio =", hex(response))
        time.sleep(1)

        response = Transfer(commands[1])
        print("Recibio = ", hex(response))
        time.sleep(1)

except KeyboardInterrupt:
    spi.deinit()
    print("Finalizado")
```

Arduino:

```
#include <SPI.h>

#define LED_PIN 9

volatile byte command = 0;
volatile bool received = false;

void setup() {
  pinMode(LED_PIN, OUTPUT);

  // Configurar Arduino como esclavo
  pinMode(MISO, OUTPUT);   // El esclavo solo puede mandar por MISO
  SPCR |= _BV(SPE);        // Habilita SPI en modo esclavo
  SPI.attachInterrupt();   // Habilita interrupción SPI
}

ISR(SPI_STC_vect) {
  command = SPDR;          // Leer byte recibido
  received = true;
}

void loop() {
  if (received) {
    byte response = 0;

    if (command == 0x01) {
      digitalWrite(LED_PIN, HIGH);
      response = 0xAA; // Ack encendido
    }
```

```
    else if (command == 0x00) {
      digitalWrite(LED_PIN, LOW);
      response = 0x55; // Ack apagado
    }
    else {
      response = 0xFF; // Comando inválido
    }

    SPDR = response;   // Cargar respuesta → Maestro la recibe en el próximo ciclo
    received = false;
  }
}
```

A-C. Punto 3: I²C con potenciómetro y LEDs

Raspberry Pi (MicroPython):

```
from machine import Pin, I2C
from utime import sleep
import time

Led = Pin("LED", Pin.OUT)
Led.on()
i2c = I2C(0, scl = Pin(1), sda= Pin(0), freq = 100000)
print("Escaneando...")
print([hex(x) for x in i2c.scan()])
time.sleep(5)
ARDUINO_ADDR = 0x08

Leds = [Pin(18, Pin.OUT), Pin(19, Pin.OUT), Pin(20, Pin.OUT)]

def LedsBinario(valor):
    for i in range(len(Leds)):
        Leds[i].value((valor >> i) & 1)

print("Program Started...")

while True:
    try:
        data = i2c.readfrom(ARDUINO_ADDR, 1)
        valor = data[0]
        print("Dato Recibido: ", valor)

        LedsBinario(valor)

        time.sleep(0.5)
        Led.toggle()

    except KeyboardInterrupt:
        break
    except Exception as e:
        print("Error I2C: ", e)
        time.sleep(1)

Led.off()
print("Finished.")
```

Arduino:

```
#include <Wire.h>

volatile byte ultimoDato = 0;   // compartido entre loop() e ISR

void setup() {
  Serial.begin(115200);

  Wire.begin(0x08);             // Esclavo I2C en 0x08
  Wire.onRequest(ReadValue);    // Callback rápido
}

void loop() {
  int valor = analogRead(A0);                   // 0..1023
  byte dato = (byte)map(valor, 0, 1023, 0, 7);  // 0..7 para 3 LEDs
  ultimoDato = dato;                            // actualizar valor compartido

  static unsigned long t0 = 0;
```

```
  if (millis() - t0 > 500) {
    t0 = millis();
    Serial.print(F("Ultimo dato: "));
    Serial.println(ultimoDato);
  }
}

void ReadValue() {
  Wire.write(ultimoDato);
}
```