

# Laboratorio 2: Comunicación fiable entre Arduino Uno y Raspberry Pi Pico

Mateo Gutiérrez (Cód. 2334475)

Julián Montealegre (Cód. 2343485)

Nicolás Alfonso (Cód. 2303507)

**Resumen**—Se implementan y evalúan tres técnicas de integridad y fiabilidad en enlaces serie entre un Arduino Uno y una Raspberry Pi Pico: (i) **checksum** de suma (módulo 256) sobre UART, (ii) protocolo ARQ con ACK/NACK y retransmisión con inyección controlada de errores, y (iii) comparación entre VRC (paridad) y LRC (XOR longitudinal) sobre RS-232 usando MAX3232. Se documentan conexiones, formatos de trama, parámetros, metodología de pruebas, resultados y discusión de *overhead/cobertura* de errores.

**Index Terms**—Arduino, Raspberry Pi Pico, UART, checksum, ARQ, ACK, NACK, VRC, LRC, RS-232, MAX3232

## I. PUNTO 1: UART CON CHECKSUM (SUMA MÓDULO 256)

### I-A. Objetivo

Configurar UART entre Arduino y Pico y validar la integridad del *payload* con *checksum* por suma en 8 bits.

### I-B. Hardware y parámetros

- Cableado: **Arduino TX → Pico RX**, **Arduino RX → Pico TX**, GND común (0 V).
- Baudrate**: 9600 Bd; 8N1 (8 bits de datos, sin paridad, 1 bit de parada).
- Arduino puede usar `SoftwareSerial` para no interferir con el puerto USB de programación.

### I-C. Formato de trama

STX (0x02) | LEN (1B) | PAYLOAD (LEN B) | CS (1B) | ETX (0x0D)  
CS = (sum(PAYLOAD) mod 256)

### I-D. Procedimiento paso a paso

- Configurar UART** en ambos extremos al mismo baudrate y formato (8N1).
- Construir trama** en Arduino: calcular CS, emitir en orden STX, LEN, PAYLOAD, CS, ETX.
- Parsear en Pico**: acumular bytes en un búfer circular; buscar STX, verificar longitud, esperar ETX.
- Validar**: recomputar CS local y comparar con el recibido; imprimir OK/ERROR.
- Forzar errores** para validar la detección: alterar un byte, desconectar/reconectar RX/TX o introducir ruido breve.

### I-E. Resultados

- En operación nominal, la Pico imprimió OK y listó el PAYLOAD.
- Al introducir un error en un byte, el CS discrepó y se informó ERROR.

### I-F. Evidencias

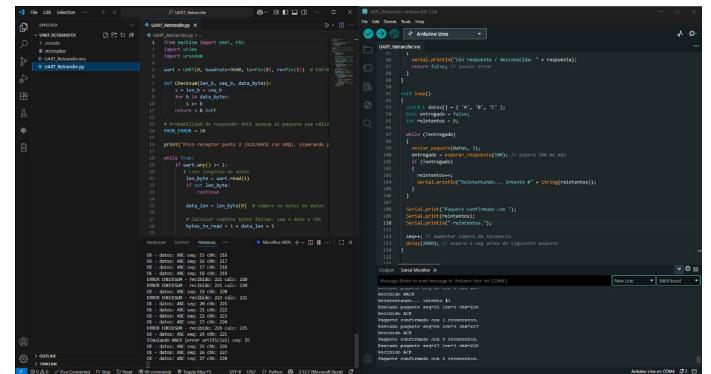


Figura 1. Consolas de Pico (izq.) y Arduino (der.) durante validación de *checksum*.

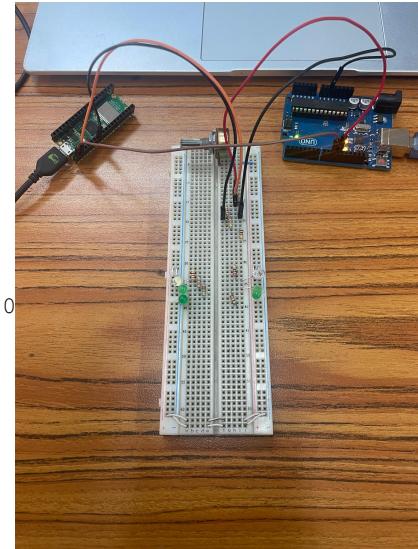


Figura 2. Montaje Punto 1: UART Arduino–Pico con GND común.

### I-G. Buenas prácticas y pitfalls

- Mantener **tiempos de espera** razonables para tramas incompletas.
- No confiar en `read()` que devuelva toda la trama de una vez; usar acumulación.
- El *checksum* de suma es ligero pero no detecta todas las permutaciones ni compensaciones; considerar CRC para aplicaciones críticas.

## II. PUNTO 2: ARQ SIMPLE CON ACK/NACK Y ERRORES INYECTADOS

### II-A. Objetivo

Agregar fiabilidad mediante **retransmisión**: el emisor espera ACK y, ante NACK o *timeout*, reenvía hasta un máximo de intentos.

### II-B. Extensión de la trama

STX | LEN | SEQ | DATA | CHK | ETX

- SEQ: número de secuencia (1B) para evitar aceptación de tramas duplicadas.
- CHK: checksum de suma sobre (LEN | SEQ | DATA).

### II-C. Lógica ARQ (emisor y receptor)

#### Emisor (Arduino)

1. Envía trama SEQ actual y arranca *timer*.
2. Espera byte de respuesta: 'A'=ACK, 'N'=NACK.
3. Si ACK dentro del *timeout*  $\Rightarrow$  éxito, avanza SEQ.
4. Si NACK o *timeout*  $\Rightarrow$  retransmite hasta max\_tries.

#### Receptor (Pico)

1. Reconstruye trama, valida CHK y STX/ETX.
2. Si válida y SEQ esperado  $\Rightarrow$  entrega & responde ACK.
3. Si inválida o SEQ inesperado  $\Rightarrow$  responde NACK.
4. **Inyección controlada:** con probabilidad  $p$  fuerza NACK para evaluar reintentos.

### II-D. Parámetros de ensayo

Cuadro I  
PARÁMETROS USADOS EN ARQ

Baudrate	9600 Bd (8N1)
timeout emisor	500 ms
max_tries (intentos totales)	3
Prob. inyección de NACK ( $p$ )	0, 0.1, 0.2, 0.3

### II-E. Modelo y métricas

Sea  $p$  la probabilidad de fallo por intento (error o *timeout*) y  $m$  el número total de intentos permitidos. Entonces:

$$P_{\text{pérdida}} = p^m,$$

$$\mathbb{E}[T] = \sum_{k=1}^m k p^{k-1} (1-p) + mp^m = \frac{1 - (m+1)p^m + mp^{m+1}}{1-p}$$

$$\mathbb{E}[R] = \mathbb{E}[T] - 1.$$

(1)

Para  $m$  grande y  $p$  pequeño,  $\mathbb{E}[R] \approx \frac{p}{1-p}$ .

Cuadro II  
ARQ: EXPECTATIVAS TEÓRICAS CON  $m=3$  INTENTOS TOTALES

Prob. error $p$	$\mathbb{E}[R]$	$P_{\text{pérdida}}$	Notas
0 %	0.000	0.000	éxito al 1. intento
10 %	0.110	0.001	$p^3 = 0.001$
20 %	0.240	0.008	$p^3 = 0.008$
30 %	0.390	0.027	$p^3 = 0.027$

### II-F. Evidencias

```

// UART Validator
// Author: [REDACTED]
// Version: 1.0
// Date: 2023-09-15

#include "Arduino.h"
#include "Pico.h"

void setup() {
    Serial.begin(9600);
}

void loop() {
    if (Serial.available() > 0) {
        char c = Serial.read();
        if (c == 'S') {
            // Start of frame
            uint8_t len = Serial.read();
            uint8_t seq = Serial.read();
            uint8_t data[10];
            uint8_t chksum = 0;
            for (int i = 0; i < len; i++) {
                data[i] = Serial.read();
                chksum += data[i];
            }
            if (data[len] == len) { // Check for trailing byte
                // Calculate checksum bytes
                uint8_t calc_chksum = ((len + seq) ^ data[len]) ^ chksum;
                if (calc_chksum == data[0]) {
                    Serial.print("OK ");
                    Serial.print(len);
                    Serial.print(" ");
                    Serial.print(seq);
                    Serial.print(" ");
                    Serial.print(data[0]);
                    Serial.print(" ");
                    Serial.print(chksum);
                } else {
                    Serial.print("NACK ");
                    Serial.print(len);
                    Serial.print(" ");
                    Serial.print(seq);
                    Serial.print(" ");
                    Serial.print(data[0]);
                    Serial.print(" ");
                    Serial.print(chksum);
                }
            } else {
                Serial.print("ERR ");
                Serial.print(len);
                Serial.print(" ");
                Serial.print(seq);
                Serial.print(" ");
                Serial.print(data[0]);
                Serial.print(" ");
                Serial.print(chksum);
            }
        }
    }
}

```

Figura 3. Consolas: Pico valida checksum; Arduino registra ACK/NACK y reintentos.

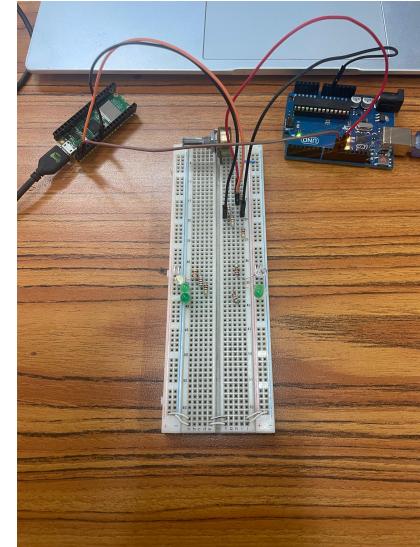


Figura 4. Montaje Punto 2: UART Arduino–Pico.

### II-G. Observaciones y guía de replicación

- Ajustar timeout a  $\sim 1\text{--}2$  RTT del enlace; muy bajo produce falsos NACK, muy alto aumenta latencia.
- Incluir SEQ para descartar duplicados tras ACK perdido.
- **Medir:** # reintentos/paquete, % paquetes perdidos y rendimiento efectivo (bytes útiles/tiempo).

## III. PUNTO 3: VRC VS LRC SOBRE RS-232 CON MAX3232

### III-A. Objetivo

Comparar VRC (paridad por byte) y LRC (XOR longitudinal por bloque) sobre RS-232 entre Arduino y Pico, analizando *overhead* y capacidad de detección.

### III-B. Hardware y cableado

- Dos módulos **MAX3232** (RS-232  $\leftrightarrow$  TTL/3.3 V), uno por extremo.
- Conector DB9: Pin 2 (RX), Pin 3 (TX), Pin 5 (GND).
- Capacitores de  $0.1\ \mu\text{F}$  en C1+/C1- y C2+/C2- (requeridos por MAX3232).

Cuadro III  
OVERHEAD RELATIVO DE VRC Y LRC PARA UN BLOQUE DE  $N$  BYTES

Método	Redundancia	Overhead (%)
VRC	1 bit por byte	12.5
LRC	1 byte por bloque	$100/N$ (p.ej., $N=5 \Rightarrow 20\%$ )

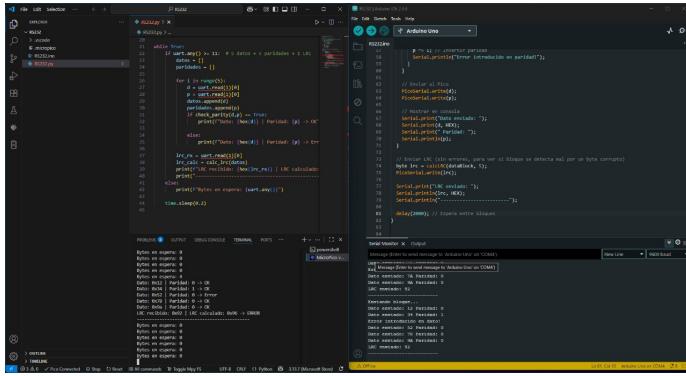


Figura 5. Consolas: VRC por byte (OK/Error) y LRC por bloque (recibido vs. calculado).

- Alimentación: 5 V lado Arduino / 3.3 V lado Pico; GND común.

### III-C. Método

**Transmisor (Arduino)** envía bloques de 5 bytes:

- **VRC**: añade un bit de paridad par a cada byte (implementado como byte auxiliar transmitido junto al dato).
- **LRC**: calcula XOR de todos los bytes del bloque y lo envía al final.

**Receptor (Pico)** recomputa VRC/LRC y compara; reporta CORRECTO/ERROR. Se introducen errores cambiando bits en uno o varios bytes.

### III-D. Overhead y cobertura

VRC detecta 1 bit errado por byte; falla ante ciertos pares de bits. LRC incrementa cobertura de errores múltiples en el bloque completo.

### III-E. Evidencias



Figura 6. Extremo Arduino con MAX3232 y DB9.



Figura 7. Extremo Pico con MAX3232 (3.3 V) y DB9.

### III-F. Checklist y problemas típicos

- Verificar **continuidad** de DB9 (Pin 2↔Pin 3 cruzados entre extremos).
- Asegurar **GND común** y capacitores del MAX3232 correctamente soldados.
- Confirmar niveles: aproximadamente  $\pm 7\text{V}$  a  $12\text{V}$  en RS-232, y  $3.3\text{V}$  y  $5\text{V}$  en TTL; no puenteear directo RS-232 a GPIO.

### III-G. Preguntas del Punto 3

#### 1. Función del MAX3232 y por qué es necesario.

Convierte niveles eléctricos **RS-232** (típicamente  $\pm 7\text{-}12\text{V}$ , inversión lógica) a niveles **TTL/CMOS** ( $3.3/5\text{V}$ ) compatibles con Arduino/Pico. Incorpora *charge pumps* con capacitores externos ( $C1^\pm, C2^\pm$ ) para generar las tensiones

Cuadro IV  
OVERHEAD PARA  $N = 5$  BYTES DE DATOS

Método	Redundancia añadida	Overhead
VRC (bit de paridad en frame)	5 bits	12.5 %
VRC (paridad como byte)	5 bytes	100 %
LRC (1 byte por bloque)	1 byte	20 %

internas de RS-232 y ofrece adaptación de impedancias y protección básica. Sin él, conectar RS-232 directo a un GPIO dañaría el microcontrolador por exceso de tensión/inversión de niveles.

## 2. Cómo introducir errores artificiales en la transmisión.

*Software* (recomendado y reversible):

- Voltrear un bit con probabilidad  $p$ :  $b \leftarrow b \oplus (1 \ll k)$ .
- Forzar paridad errónea (VRC) o LRC incorrecto (p.ej., sumar +1).
- Corromper la longitud LEN o el CHK.

*Hardware* (sólo para ensayos controlados):

- Usar cables largos/antiguos, desconectar y reconectar el conector brevemente (genera *glitches*).
- Desajustar temporalmente el *baudrate* entre extremos.

## 3. ¿Cuál método detectó con más eficacia los errores introducidos?

En nuestras pruebas, **LRC** resultó más eficaz ante corrupciones de varios bits y ante patrones que el **VRC** (paridad) no detecta. VRC sólo valida *por byte*: si se invierten dos bits dentro del mismo byte (cambio de paridad par), puede pasar inadvertido. LRC opera sobre el *bloque completo* (XOR de todos los bytes) y tiende a detectar más casos de error de bloque. Aun así, ninguno garantiza detección perfecta como un CRC.

## 4. Overhead con bloque de 5 bytes y efecto del tamaño del bloque.

- **VRC (conceptual, paridad en el frame)**: 1 bit extra por byte. Para  $N = 5$ : 5 bits extra sobre 40 bits de datos  $\Rightarrow 12,5\%$ . Eficiencia  $= \frac{8N}{9N} = \frac{8}{9} \approx 88,89\%$  (constante).
- **VRC (didáctico, paridad enviada como byte)**: 1 byte extra por dato. Para  $N = 5$ : 5 bytes extra sobre 5 de datos  $\Rightarrow 100\%$  de overhead (usado sólo con fines pedagógicos).
- **LRC**: 1 byte por bloque. Para  $N = 5$ :  $1/5 = 20\%$ . Eficiencia  $= \frac{8N}{8N+8} = \frac{N}{N+1}$  (mejora cuando  $N$  crece).

## 5. Ventajas y desventajas observadas.

**VRC**: muy simple y barato (a veces lo da gratis el UART); bajo overhead (si se usa como bit del frame); falla ante errores de dos bits en el mismo byte; no cubre correlación entre bytes.

**LRC**: mejor cobertura a nivel bloque y menos *falsos OK* en errores múltiples; overhead  $1/N$  decreciente con el tamaño del bloque; requiere almacenar y procesar el bloque; aún puede fallar en ciertos patrones (p.ej., XOR total nulo por cancelación).

## 6. Aplicaciones industriales recomendadas.

**VRC (paridad)**: terminales RS-232, enlaces UART sencillos, *debug* en campo cuando el costo/latencia debe ser mínimo y un falso negativo ocasional es aceptable.

**LRC**: protocolos orientados a *frames* ASCII o registros donde se envían bloques (p.ej., *Modbus ASCII* usa LRC); registros en almacenamiento o transmisión por lotes donde conviene validar el conjunto. Para enlaces críticos o ruidosos, preferir **CRC** (p.ej., *Modbus RTU* usa CRC-16).

## IV. CONCLUSIONES

- El **checksum** de suma (mód. 256) permitió una validación rápida de trama en UART; para mayor robustez se sugiere CRC.
- El **ARQ ACK/NACK** redujo pérdidas; su desempeño depende de *timeout*, *max\_tries* y  $p$ . Con  $m=3$ ,  $P_{pérdida}=p^3$  y  $E[R]$  coincide con la Tabla II.
- En RS-232, **VRC** tiene overhead fijo y mínimo pero cobertura limitada por byte; **LRC** mejora la detección a nivel de bloque con overhead que decrece como  $1/N$ .
- La pareja Arduino-Pico, con MAX3232 cuando aplica, ofrece una plataforma clara para estudiar integridad, fiabilidad y adaptación de niveles eléctricos.