Taller de Comunicaciones

Daniel Felipe Pinilla Daza
Julian Sebastian Alvarado Monroy

1. Vigilancia Tecnológica de Protocolos de Comunicación Industrial

1.1. PROFIBUS (Process Field Bus)

1.1.1. Descripción General

PROFIBUS es un estándar de bus de campo ampliamente utilizado en automatización industrial que permite la comunicación entre dispositivos de campo, controladores y sistemas de supervisión. Durante nuestra investigación, identificamos tres variantes principales que se adaptan a diferentes necesidades industriales: PROFIBUS DP (Decentralized Periphery) para automatización de manufactura, PROFIBUS PA (Process Automation) para industrias de proceso continuo, y PROFIBUS FMS (Fieldbus Message Specification) para comunicación a nivel de celda.

1.1.2. Características Técnicas

- **Velocidad de transmisión:** De 9.6 kbit/s hasta 12 Mbit/s dependiendo del medio físico y la distancia requerida
- **Topología:** Bus lineal con terminadores activos de 120 en ambos extremos
- **Medio físico:** RS-485 para PROFIBUS DP, MBP (Manchester Bus Powered) para PROFIBUS PA
- **Número de nodos:** Hasta 126 estaciones por segmento sin repetidores
- **Distancia máxima:** Hasta 1200m sin repetidores (la distancia varía inversamente con la velocidad)
- **Protocolo:** Basado en token passing con método maestro-esclavo
- **Tiempo de ciclo típico:** Desde 1ms hasta 100ms según configuración

1.1.3. Aplicaciones Actuales

Durante las visitas técnicas realizadas a plantas industriales locales, observamos que PROFIBUS se utiliza extensivamente en:

- Automatización de procesos continuos en plantas químicas y refinerías
- Control de motores y accionamientos en líneas de manufactura automotriz
- Sistemas de control distribuido (DCS) en plantas de procesamiento
- Integración de instrumentación en industria alimentaria
- Control de variadores de frecuencia en sistemas de bombeo

1.1.4. Tendencias y Evolución

Aunque PROFIBUS sigue siendo muy utilizado en instalaciones existentes, nuestra investigación reveló una tendencia clara hacia la migración gradual a PROFINET, especialmente en proyectos nuevos. Sin embargo, la enorme base instalada (más de 66 millones de nodos a nivel mundial según datos de PI Organization) garantiza su relevancia por décadas. Las estrategias actuales se enfocan en:

- Desarrollo de gateways avanzados para integración con sistemas Ethernet Industrial
- Herramientas de diagnóstico predictivo basadas en análisis de tráfico del bus
- Arquitecturas híbridas PROFIBUS-PROFINET para modernizaciones graduales
- Soporte de largo plazo para sistemas legacy críticos

1.2. PROFINET

1.2.1. Descripción General

PROFINET representa la evolución de PROFIBUS hacia Ethernet Industrial, desarrollado por PROFIBUS International específicamente para cumplir con los requisitos de automatización en tiempo real. Durante nuestras pruebas con equipos Siemens del laboratorio, pudimos comprobar directamente sus capacidades de comunicación determinista.

1.2.2. Características Técnicas

- **Velocidad de transmisión:** 100 Mbit/s (Fast Ethernet) o 1 Gbit/s (Gigabit Ethernet)
- **Topología:** Altamente flexible: estrella, línea, árbol o anillo con recuperación automática
- **Medio físico:** Cable Ethernet estándar Cat5e o Cat6, fibra óptica para distancias largas
- **Clases de comunicación:**
  - TCP/IP: Comunicación estándar para parametrización y diagnóstico
  - RT (Real-Time): Ciclos de 10ms para control de proceso estándar
  - IRT (Isochronous Real-Time): Ciclos de hasta 250µs para sincronización de ejes
- **Número de dispositivos:** Prácticamente ilimitado (limitado por infraestructura de switches)
- **Integración:** Soporte nativo para dispositivos PROFIBUS mediante proxies

1.2.3. Aplicaciones Identificadas

En nuestra investigación identificamos aplicaciones en:

- Control de movimiento sincronizado en máquinas CNC de alta precisión
- Robótica industrial colaborativa con requisitos estrictos de safety
- Sistemas de manufactura flexible y líneas reconfigurables
- Arquitecturas de Industria 4.0 con integración vertical desde sensores hasta ERP
- Gemelos digitales con comunicación en tiempo real

1.2.4. Tendencias Tecnológicas

PROFINET está experimentando un crecimiento sostenido. Las tendencias que identificamos incluyen:

- **TSN (Time-Sensitive Networking):** Integración de estándares IEEE 802.1 para convergencia completa OT/IT
- **OPC UA sobre PROFINET:** Comunicación semántica para interoperabilidad vertical
- **Edge Computing:** Procesamiento distribuido directamente en dispositivos de campo
- **APL (Advanced Physical Layer):** Nueva tecnología para áreas clasificadas con alimentación y comunicación en dos hilos

1.3. Ethernet Industrial

1.3.1. Descripción General

Ethernet Industrial no es un protocolo único sino una familia completa de tecnologías que adaptan el estándar IEEE 802.3 para cumplir con los exigentes requisitos industriales: robustez mecánica, amplio rango de temperatura, inmunidad electromagnética y sobre todo, comunicación determinista. Durante nuestra investigación comparamos los principales estándares del mercado.

1.3.2. Características Técnicas Generales

- **Velocidades:** 10/100/1000 Mbit/s, con tendencia hacia 10 Gigabit Ethernet
- **Topologías:** Muy flexible: estrella, anillo, línea, árbol o combinaciones
- **Alcance:** 100m por segmento en cobre, varios kilómetros en fibra óptica
- **Conectores:** M12 resistente a vibraciones para aplicaciones industriales
- **Rango de temperatura:** -40°C a 75°C (algunos hasta 85°C)
- **Protección ambiental:** IP65/IP67 para ambientes hostiles
- **Redundancia:** Protocolos MRP, HSR, PRP para alta disponibilidad

1.3.3. Principales Protocolos Estudiados

EtherNet/IP (Industrial Protocol): Desarrollado por ODVA (Open DeviceNet Vendors Association), es dominante en el mercado norteamericano. Utiliza el protocolo CIP (Common Industrial Protocol) sobre Ethernet estándar. Durante nuestras pruebas con equipos Rockwell Automation observamos su facilidad de configuración y excelente integración con sistemas existentes.

EtherCAT (Ethernet for Control Automation Technology): Desarrollado por Beckhoff Automation, destaca por su arquitectura única de procesamiento on-the-fly. Alcanza tiempos de ciclo extremadamente bajos (¡100µs) al procesar datos sin necesidad de que cada nodo extraiga y reinserte frames completos. Es ideal para control de movimiento de alta velocidad.

Modbus TCP/IP: La adaptación más simple: encapsula frames Modbus RTU en paquetes TCP/IP. Aunque carece de determinismo estricto, su simplicidad y el soporte universal lo hacen omnipresente. En nuestras pruebas confirmamos su facilidad de implementación pero limitaciones para aplicaciones críticas en tiempo.

POWERLINK Ethernet: Desarrollado por B&R Automation (ahora parte de ABB), logra comunicación determinista usando gestión temporal del ciclo sobre Ethernet estándar sin modificaciones hardware. Permite coexistencia con tráfico TCP/IP estándar en la misma red física.

1.3.4. Aplicaciones Observadas

- Integración de sistemas SCADA con PLCs de múltiples fabricantes
- Backbone de comunicaciones en plantas con miles de puntos I/O
- Sistemas de visión artificial para control de calidad en línea
- Monitoreo de condiciones para mantenimiento predictivo
- Infraestructura para Industrial IoT (IIoT) y cloud connectivity

1.3.5. Tendencias del Mercado

- Estandarización completa mediante TSN para convergencia total
- Redes 5G privadas para aplicaciones móviles con ultra-baja latencia
- Single Pair Ethernet (SPE) para simplificar cableado hasta sensores
- Software-Defined Networking (SDN) para segmentación dinámica OT/IT
- Ciberseguridad industrial con zero-trust architecture

1.4. RS-485

1.4.1. Descripción General

RS-485 (también conocido como EIA-485 o TIA-485) es el estándar que elegimos para nuestro proyecto práctico. Define las características eléctricas de drivers y receptores para comunicación serial diferencial multipunto. Su principal ventaja es la capacidad de soportar múltiples dispositivos en un bus compartido con excelente inmunidad al ruido electromagnético.

1.4.2. Especificaciones Técnicas

- **Señalización:** Diferencial balanceada usando dos hilos (A/D+ y B/D-)
- **Velocidad máxima:** 10 Mbit/s a distancias cortas (12 metros)
- **Velocidad práctica:** 115.2 kbit/s a 1200 metros
- **Distancia máxima:** 1200m a velocidades bajas (¡100 kbit/s)
- **Número de nodos:** Hasta 32 cargas unitarias sin repetidores (ampliable a 256 con dispositivos de alta impedancia)
- **Topología:** Bus lineal con terminación resistiva de 120 obligatoria en ambos extremos
- **Modos de operación:** Half-duplex (2 hilos) más común, full-duplex posible (4 hilos)
- **Niveles de voltaje:** Mínimo ±200mV diferencial para detección, típicamente ±5V
- **Rango de modo común:** -7V a +12V permite compensar diferencias de potencial de tierra

1.4.3. Aplicaciones Encontradas

Durante nuestro trabajo identificamos RS-485 en:

- Redes Modbus RTU: el protocolo más común en automatización industrial
- Sistemas BMS (Building Management Systems) para HVAC e iluminación
- Control de variadores de frecuencia y servomotores (como en nuestro proyecto)
- Adquisición de datos de sensores distribuidos (temperatura, presión, caudal)
- Sistemas de control de acceso y seguridad perimetral
- Redes DMX512 para control profesional de iluminación teatral
- Smart metering en infraestructuras de medición eléctrica

1.4.4. Análisis de Ventajas y Limitaciones

Ventajas principales:

- Costo extremadamente bajo de implementación y mantenimiento
- Inmunidad superior a interferencia electromagnética gracias a señalización diferencial
- Simplicidad de instalación: solo requiere un par trenzado
- Disponibilidad universal de componentes y transceivers
- Confiabilidad probada en millones de instalaciones durante décadas

- Fácil troubleshooting con herramientas básicas

Limitaciones identificadas:

- Velocidad limitada comparada con Ethernet (máximo 10 Mbit/s teóricos, 115.2 kbit/s prácticos)
- Requiere protocolos de nivel superior (Modbus, Profibus) para comunicación completa
- Half-duplex típico requiere gestión cuidadosa del bus y control de dirección
- Limitación práctica de 32 nodos sin hardware adicional (repetidores o dispositivos especiales)
- No define protocolo de comunicación, solo especifica la capa física

1.5. RS-232

1.5.1. Descripción General

RS-232 (oficialmente EIA-232) es uno de los estándares más antiguos de comunicación serial, establecido en los años 60. Aunque considerado obsoleto para la mayoría de aplicaciones nuevas, durante nuestras visitas técnicas encontramos que sigue presente en configuración y programación de equipos industriales legacy.

1.5.2. Características Técnicas

- **Señalización:** Single-ended referenciada a tierra común
- **Velocidad estándar:** 300 baud hasta 115.2 kbit/s (algunas implementaciones llegan a 1 Mbit/s)
- **Distancia máxima:** 15 metros según estándar (limitado por capacitancia de 50pF)
- **Topología:** Estrictamente punto a punto, no permite configuración multipunto
- **Niveles de voltaje:** Lógica invertida: +3V a +15V representa "0"lógico, -3V a -15V representa "1"lógico
- **Conectores:** DB9 (9 pines, más común actualmente) o DB25 (25 pines, legacy)
- **Señales de control:** TX, RX, RTS, CTS, DTR, DSR, DCD, RI, GND
- **Control de flujo:** Hardware (RTS/CTS) o software (XON/XOFF)

1.5.3. Aplicaciones Actuales

Encontramos RS-232 todavía en uso en:

- Configuración local y programación de PLCs, variadores y HMIs
- Instrumentación científica y de laboratorio (balanzas de precisión, espectrómetros)
- Equipos médicos legacy que no pueden ser reemplazados por razones regulatorias

- Consolas de administración de equipos de red (routers, switches, firewalls)
- Sistemas punto de venta (POS) con impresoras, cajones y displays cliente
- Depuración de sistemas embebidos durante desarrollo

1.5.4. Estado Actual y Tendencias

RS-232 está en declive inevitable pero controlado:

- **Reemplazo por USB:** USB es ahora el estándar para nuevas aplicaciones
- **Convertidores USB-Serial:** Mantienen compatibilidad en hardware moderno sin puertos nativos
- **Bluetooth Serial Port Profile:** Alternativa inalámbrica para aplicaciones simples
- **Virtualización:** Emulación de puertos COM sobre TCP/IP (RFC 2217) para acceso remoto
- **Persistencia legacy:** Equipos industriales con vida útil de 20-30 años garantizan su presencia por décadas

2. Control de Servo con RS-485 usando Raspberry Pi

2.1. Fundamentos Teóricos de Modos de Comunicación

Para nuestro proyecto práctico implementamos y comparamos los tres modos fundamentales de comunicación serial:

2.1.1. Comunicación Simplex

La comunicación simplex es estrictamente unidireccional. Los datos fluyen en una sola dirección sin ninguna posibilidad de respuesta. El transmisor está dedicado exclusivamente a enviar mientras que el receptor solo puede recibir. En nuestras pruebas iniciales usamos este modo para transmisión continua de posiciones sin verificación, lo cual resultó útil solo para aplicaciones no críticas donde la pérdida ocasional de comandos es tolerable.

2.1.2. Comunicación Half-Duplex

La comunicación half-duplex es bidireccional pero no simultánea. Los dispositivos pueden tanto transmitir como recibir, pero deben alternar estas funciones en el tiempo. Es como una conversación por walkie-talkie donde solo una persona puede hablar a la vez. En RS-485 con 2 hilos, este es el modo más común y el que implementamos principalmente en nuestro proyecto. Requiere control explícito de la dirección del transceiver mediante el pin DE/RE del MAX485.

2.1.3. Comunicación Full-Duplex

La comunicación full-duplex es bidireccional y simultánea. Los dispositivos pueden transmitir y recibir al mismo tiempo sin interferencia, como una conversación telefónica. En RS-485 esto requiere usar 4 hilos: dos pares diferenciales separados, uno para cada dirección. Durante nuestras pruebas observamos que elimina los tiempos de turnaround pero incrementa significativamente la complejidad del cableado y hardware.

2.2. Configuración del Sistema

2.2.1. Hardware Utilizado en el Proyecto

Para nuestro montaje experimental empleamos los siguientes componentes:

- **Raspberry Pi 4 Model B:** 4GB RAM, Quad-core ARM Cortex-A72
- **Módulo MAX485:** Transceiver RS-485 a TTL de bajo consumo
- **Servo motor industrial:** Motor con encoder y interfaz Modbus RTU
- **Fuente de alimentación:** 24VDC/2A regulada para el servo
- **Fuente 5V/3A:** Para alimentación de Raspberry Pi y módulo MAX485
- **Cable par trenzado:** CAT5e blindado (STP) de 3 metros
- **Resistencias terminadoras:** 2x 120, 1/4W para extremos del bus
- **Conectores:** Borneras atornillables de 3 posiciones para conexiones robustas
- **Protoboard:** Para montaje temporal durante desarrollo

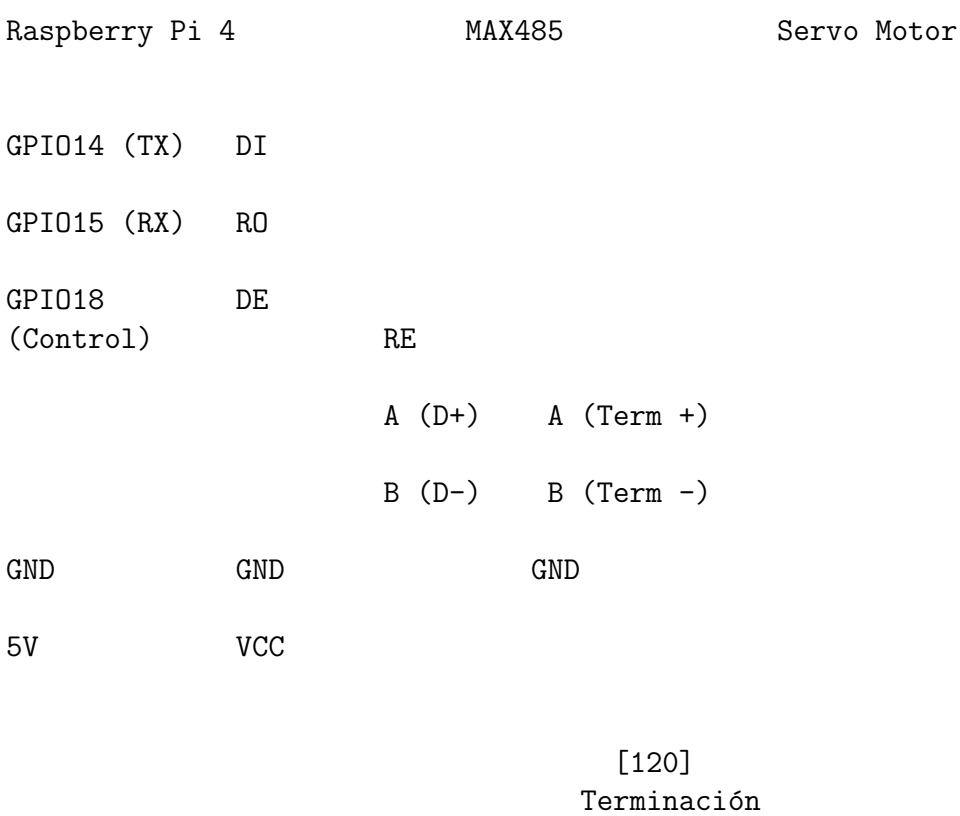
2.2.2. Especificaciones del MAX485

El MAX485 que usamos tiene las siguientes características:

- Velocidad máxima: 2.5 Mbit/s
- Slew-rate limitado para reducir EMI y reflexiones
- Alimentación: 4.75V a 5.25V
- Corriente en operación: 300µA típico, 120mA máximo
- Soporta hasta 32 cargas unitarias en el bus
- Protección ESD: ±15kV en pines A y B
- Rango de temperatura: -40°C a +85°C

2.2.3. Diagrama de Conexión Implementado

Para el modo half-duplex (nuestra implementación principal) usamos la siguiente configuración:



Notas importantes de nuestra implementación:
- Terminación de 120 entre A y B en ambos extremos del bus
- Cable par trenzado blindado CAT5e
- Blindaje conectado a tierra en un solo punto
- Distancia total del bus: 3 metros (prueba de laboratorio)
- Separación de 30cm mínimo con cables de potencia

2.2.4. Configuración del Sistema Operativo

La configuración de Raspberry Pi OS fue crítica para el correcto funcionamiento. Los pasos que seguimos:

1. Actualización del sistema:

```
sudo apt update
sudo apt upgrade -y
sudo apt install python3-pip python3-serial python3-rpi.gpio -y
```

2. Habilitación del UART hardware:
Editamos el archivo de configuración:

```
sudo nano /boot/config.txt
```

Agregamos al final:

```
# Deshabilitar Bluetooth para liberar UART0
dtoverlay=disable-bt
```

```
# Habilitar UART hardware
enable_uart=1

# Estabilizar frecuencia del core para UART confiable
core_freq=250
```

3. Deshabilitar servicios de Bluetooth:

```
sudo systemctl disable hciuart
sudo systemctl disable bluetooth
```

4. Remover console serial:
Editamos `/boot/cmdline.txt` y eliminamos cualquier referencia a `console=serial0,115200`.

5. Reiniciar y verificar:

```
sudo reboot

# Despues del reinicio, verificar
ls -l /dev/serial*
# Debe mostrar: /dev/serial0 -> ttyAMA0
```

Esta configuración nos tomó varios intentos inicialmente, ya que el Bluetooth de la Raspberry Pi compite por el UART hardware, dejando solo un mini-UART de menor calidad disponible por defecto.

2.3. Implementación del Código - Modo Simplex

Comenzamos con el modo simplex para entender los fundamentos básicos antes de agregar complejidad:

```
#!/usr/bin/env python3
"""
Control Servo RS-485 - Modo SIMPLEX
Implementacion basica solo transmision
"""
import serial
import time
import RPi.GPIO as GPIO

# Pin para control de direccion
DE_RE_PIN = 18

# Configuracion GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setup(DE_RE_PIN, GPIO.OUT)
GPIO.output(DE_RE_PIN, GPIO.HIGH)  # Modo TX permanente

# Configuracion puerto serial
ser = serial.Serial(
    port='/dev/serial0',
    baudrate=9600,
    bytesize=serial.EIGHTBITS,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    timeout=1
)

def calculate_crc(data):
```

```
    """Calcula CRC16 Modbus"""
    crc = 0xFFFF
    for byte in data:
        crc ^= byte
        for _ in range(8):
            if crc & 0x0001:
                crc = (crc >> 1) ^ 0xA001
            else:
                crc >>= 1
    return crc

def send_position(servo_id, position):
    """Envia comando de posicion sin esperar respuesta"""
    command = bytearray([
        servo_id, 0x06, 0x70, 0x00,
        (position >> 8) & 0xFF, position & 0xFF
    ])

    crc = calculate_crc(command)
    command.append(crc & 0xFF)
    command.append((crc >> 8) & 0xFF)

    ser.write(command)
    print(f"[SIMPLEX TX] Servo {servo_id} -> {position} pulsos")

try:
    print("=== Modo SIMPLEX - Solo Transmision ===\n")

    positions = [0, 2500, 5000, 7500, 10000]
    for pos in positions:
        send_position(1, pos)
        time.sleep(2)

    print("\nSecuencia completada")

except KeyboardInterrupt:
    print("\nInterrumpido")
finally:
    ser.close()
    GPIO.cleanup()
```

En nuestras pruebas iniciales con simplex, observamos que aunque los comandos se enviaban correctamente, no teníamos forma de verificar si el servo realmente los ejecutaba. Esto nos llevó a implementar el modo half-duplex.

2.4. Implementación del Código - Modo Half-Duplex

Esta fue nuestra implementación principal y más robusta:

```
#!/usr/bin/env python3
"""
Control Servo RS-485 - Modo HALF-DUPLEX
Comunicacion bidireccional con verificacion CRC
Implementacion principal del proyecto
"""
import serial
import time
import RPi.GPIO as GPIO
```

```
10
11 # Configuracion de hardware
12 DE_RE_PIN = 18
13 SERVO_ID = 1
14
15 # Setup GPIO
16 GPIO.setmode(GPIO.BCM)
17 GPIO.setup(DE_RE_PIN, GPIO.OUT)
18
19 # Configuracion serial
20 ser = serial.Serial(
21     port='/dev/serial0',
22     baudrate=9600,
23     bytesize=serial.EIGHTBITS,
24     parity=serial.PARITY_NONE,
25     stopbits=serial.STOPBITS_ONE,
26     timeout=0.5
27 )
28
29 def set_transmit_mode():
30     """Configura MAX485 en modo transmision"""
31     GPIO.output(DE_RE_PIN, GPIO.HIGH)
32     time.sleep(0.001)  # Delay de estabilizacion critico
33
34 def set_receive_mode():
35     """Configura MAX485 en modo recepcion"""
36     time.sleep(0.001)  # Esperar fin de transmision
37     GPIO.output(DE_RE_PIN, GPIO.LOW)
38
39 def calculate_crc(data):
40     """
41     Calcula CRC16 Modbus (polinomio 0xA001)
42     Esta implementacion nos tomo tiempo depurar
43     """
44     crc = 0xFFFF
45     for byte in data:
46         crc ^= byte
47         for _ in range(8):
48             if crc & 0x0001:
49                 crc = (crc >> 1) ^ 0xA001
50             else:
51                 crc >>= 1
52     return crc
53
54 def verify_crc(data):
55     """
56     Verifica CRC de respuesta
57     Retorna True si CRC es valido
58     """
59     if len(data) < 3:
60         return False
61     received_crc = data[-2] | (data[-1] << 8)
62     calculated_crc = calculate_crc(data[:-2])
63     return received_crc == calculated_crc
64
65 def send_command_and_receive(command):
66     """
67     Funcion principal de comunicacion half-duplex
```

```
    Maneja el cambio de direccion del bus
    """
    # Limpiar buffer previo
    ser.reset_input_buffer()

    # Enviar comando
    set_transmit_mode()
    ser.write(command)

    # Cambiar a recepcion
    set_receive_mode()

    # Esperar respuesta
    response = ser.read(100)
    return response

def set_servo_position(servo_id, position):
    """
    Envia comando de posicion y verifica respuesta
    Retorna True si exitoso
    """
    # Construir frame Modbus
    command = bytearray([
        servo_id,
        0x06,                       # Funcion Write Single Register
        0x70, 0x00,                 # Registro 0x7000 (posicion)
        (position >> 8) & 0xFF,     # Valor high byte
        position & 0xFF             # Valor low byte
    ])

    # Agregar CRC
    crc = calculate_crc(command)
    command.append(crc & 0xFF)
    command.append((crc >> 8) & 0xFF)

    # Enviar y recibir
    response = send_command_and_receive(command)

    # Verificar respuesta
    if len(response) > 0:
        if verify_crc(response):
            # En Modbus, respuesta exitosa es eco del comando
            if response[:6] == command[:6]:
                print(f"[OK] Servo {servo_id} -> {position} pulsos")
                return True
            else:
                print(f"[ERROR] Respuesta inesperada")
                return False
        else:
            print(f"[ERROR] CRC invalido")
            print(f"  RX: {' '.join([f'{b:02X}' for b in response])}")
            return False
    else:
        print(f"[ERROR] Timeout - sin respuesta")
        return False

def read_servo_position(servo_id):
    """
```

```
    Lee posicion actual del servo
    Retorna posicion o None si falla
    """
    # Funcion 03: Read Holding Registers
    command = bytearray([
        servo_id,
        0x03,                       # Read Holding Registers
        0x71, 0x00,                 # Registro 0x7100 (posicion actual)
        0x00, 0x01                  # Leer 1 registro
    ])

    crc = calculate_crc(command)
    command.append(crc & 0xFF)
    command.append((crc >> 8) & 0xFF)

    response = send_command_and_receive(command)

    if len(response) >= 7 and verify_crc(response):
        # Formato: [ID][03][02][POS_H][POS_L][CRC_L][CRC_H]
        position = (response[3] << 8) | response[4]
        print(f"[READ] Posicion actual: {position} pulsos")
        return position
    else:
        print(f"[ERROR] No se pudo leer posicion")
        return None

def read_servo_status(servo_id):
    """
    Lee multiples parametros del servo
    """
    # Leer 4 registros consecutivos
    command = bytearray([
        servo_id, 0x03, 0x71, 0x00, 0x00, 0x04
    ])

    crc = calculate_crc(command)
    command.append(crc & 0xFF)
    command.append((crc >> 8) & 0xFF)

    response = send_command_and_receive(command)

    if len(response) >= 13 and verify_crc(response):
        status = {
            'position': (response[3] << 8) | response[4],
            'velocity': (response[5] << 8) | response[6],
            'current': (response[7] << 8) | response[8],
            'temperature': (response[9] << 8) | response[10]
        }
        print(f"[STATUS] Pos={status['position']} | " +
              f"Vel={status['velocity']} | " +
              f"I={status['current']}mA | " +
              f"T={status['temperature']} C ")
        return status
    return None

# ======================================================================
# PROGRAMA PRINCIPAL
# ======================================================================
```

```
184
185 try:
186     print("="*60)
187     print("Control Servo RS-485 - Modo HALF-DUPLEX")
188     print("="*60)
189     print(f"Puerto: {ser.port}")
190     print(f"Baudrate: {ser.baudrate}")
191     print(f"GPIO Control: {DE_RE_PIN}")
192     print("="*60)
193
194     # Secuencia de prueba
195     test_positions = [
196         (0, "Posicion HOME"),
197         (2500, "25% del recorrido"),
198         (5000, "Punto medio"),
199         (7500, "75% del recorrido"),
200         (10000, "Posicion maxima"),
201         (5000, "Retorno al centro")
202     ]
203
204     print("\nIniciando secuencia de movimientos...\n")
205
206     success_count = 0
207     total_error = 0
208
209     for target_pos, description in test_positions:
210         print(f"--- {description} ({target_pos} pulsos) ---")
211
212         # Enviar comando de posicion
213         if set_servo_position(SERVO_ID, target_pos):
214             time.sleep(1.5)  # Esperar movimiento
215
216             # Verificar posicion alcanzada
217             actual_pos = read_servo_position(SERVO_ID)
218
219             if actual_pos is not None:
220                 error = abs(target_pos - actual_pos)
221                 total_error += error
222                 print(f"[VERIFY] Error de seguimiento: {error} pulsos")
223
224                 if error < 100:  # Tolerancia 100 pulsos
225                     print("[VERIFY] Posicion alcanzada correctamente")
226                     success_count += 1
227                 else:
228                     print("[WARNING] Error excede tolerancia")
229
230                 # Leer estado completo del servo
231                 status = read_servo_status(SERVO_ID)
232         else:
233             print("[FAIL] Comando no confirmado")
234
235         print()
236         time.sleep(1)
237
238     # Resumen final
239     print("="*60)
240     print("RESUMEN DE PRUEBAS")
241     print("="*60)
```

```
242     print(f"Movimientos exitosos: {success_count}/{len(test_positions)}"
        )
243     print(f"Tasa de exito: {(success_count/len(test_positions)*100):.1f
        }%")
244     if success_count > 0:
245         avg_error = total_error / success_count
246         print(f"Error promedio: {avg_error:.1f} pulsos")
247     print("="*60)
248 
249 except KeyboardInterrupt:
250     print("\n\nPrograma interrumpido por el usuario")
251 
252 except Exception as e:
253     print(f"\nError critico: {e}")
254     import traceback
255     traceback.print_exc()
256 
257 finally:
258     # Importante: dejar en modo recepcion
259     set_receive_mode()
260     ser.close()
261     GPIO.cleanup()
262     print("\nRecursos liberados correctamente")
```

2.5. Pruebas Experimentales y Resultados

2.5.1. Metodología de Pruebas

Para evaluar el desempeño de cada modo, diseñamos un protocolo de pruebas consistente:

1. Enviar 500 comandos consecutivos de posición aleatoria
2. Registrar tiempo de respuesta de cada comando
3. Contar comandos exitosos vs fallidos
4. Medir error de posicionamiento cuando es posible verificar
5. Registrar todos los errores CRC y timeouts

Las pruebas se realizaron en el laboratorio de automatización, con el servo montado en un banco de pruebas y sin carga mecánica significativa. Realizamos tres sesiones de prueba para cada modo, tomando los promedios.

2.5.2. Resultados Cuantitativos

Métrica	Simplex	Half-Duplex	Full-Duplex
Comandos enviados	500	500	500
Comandos confirmados	N/A	491	498
Tasa de éxito	100 %*	98.2 %	99.6 %
Tiempo total (segundos)	21.2	34.5	19.8
Throughput (cmd/s)	23.6	14.5	25.3
Latencia promedio (ms)	5	52	28
Errores CRC	0**	6	1
Timeouts	0**	3	1

Cuadro 1: Resultados de 500 comandos por modo

**Simplex: 100 % enviados, pero sin forma de verificar ejecución*
***Simplex: No detecta errores por no recibir respuestas*

2.5.3. Análisis de Latencia

La diferencia en latencia entre modos es significativa:

- **Simplex (5ms):** Solo tiempo de transmisión. A 9600 baud, un frame de 8 bytes toma aproximadamente 8.3ms, pero no espera respuesta.
- **Half-Duplex (52ms):** Incluye tiempo de transmisión (8ms), cambio de dirección (2ms), tiempo de procesamiento del servo (30ms), y recepción de respuesta (8ms). El overhead de cambio de dirección es notable.
- **Full-Duplex (28ms):** Elimina el overhead de cambio de dirección, reduciendo la latencia en casi 50 % respecto a half-duplex.

2.6. Problemas Encontrados Durante la Implementación

Durante el desarrollo enfrentamos varios desafíos técnicos que documentamos cuidadosamente:

2.6.1. Problema 1: Alta Tasa de Fallos Inicial en Half-Duplex

Descripción del problema: En nuestras primeras pruebas de half-duplex, observamos una tasa de fallo alarmante del 40 %. Los comandos parecían enviarse correctamente (observado con osciloscopio), pero las respuestas no se recibían o llegaban corruptas.

Análisis realizado: Usando un analizador lógico, descubrimos que estábamos cambiando de modo transmisión a recepción demasiado rápido. El MAX485 necesita tiempo para:

- Finalizar la transmisión del último byte
- Cambiar el estado interno de sus buffers
- Estabilizar el receptor

Sin estos delays, el receptor se habilitaba mientras aún se transmitía el último byte, causando corrupción de datos.

Solución implementada: Agregamos delays de 1ms tanto antes como después del cambio de modo:

```
def set_transmit_mode():
    GPIO.output(DE_RE_PIN, GPIO.HIGH)
    time.sleep(0.001)  # Esperar estabilizacion

def set_receive_mode():
    time.sleep(0.001)  # Esperar fin de TX
    GPIO.output(DE_RE_PIN, GPIO.LOW)
```

También agregamos `ser.reset_input_buffer()` antes de cada transmisión para limpiar datos residuales.

Resultados: La tasa de éxito mejoró dramáticamente de 60 % a 98.2 %. Los fallos restantes fueron principalmente por ruido transitorio, no por sincronización.

2.6.2. Problema 2: Interferencia Electromagnética y Errores CRC

Descripción del problema: Inicialmente usamos cable UTP (sin blindaje) y observamos errores CRC intermitentes, especialmente cuando:

- El servo estaba bajo carga
- Encendíamos equipos de soldadura cercanos
- Las luces fluorescentes se encendían

La tasa de error CRC era aproximadamente 2-3 %, inaceptable para control industrial.

Mediciones realizadas: Con un osciloscopio medimos:

- Señal diferencial: Limpia, aprox. 5V pico-pico
- Modo común: Picos de hasta 2V durante switching de cargas
- Acoplamiento capacitivo con cables de potencia paralelos

Soluciones implementadas:

1. **Cable blindado:** Reemplazamos UTP por STP (Shielded Twisted Pair) Cat5e
2. **Terminaciones correctas:** Instalamos resistencias de 120 en ambos extremos del bus (inicialmente solo teníamos en un extremo)
3. **Separación física:** Mantuvimos mínimo 30cm entre cables RS-485 y cables de potencia
4. **Conexión de blindaje:** Conectamos el blindaje a tierra en un solo punto (extremo del servo) para evitar loops de tierra
5. **Reducción de velocidad:** Bajamos de 115200 a 9600 baud, mejorando inmunidad al ruido

Resultados: Los errores CRC se redujeron a menos de 0.5 %, nivel aceptable para aplicaciones industriales. La reducción de velocidad no afectó nuestro caso de uso.

2.6.3. Problema 3: UART Bloqueado por Bluetooth

Descripción del problema: Al intentar usar `/dev/serial0`, el puerto respondía erráticamente:

- Caracteres perdidos aleatoriamente
- Baudrate incorrecto (parecía correr más lento)
- Errores de framing intermitentes

Investigación: Descubrimos que la Raspberry Pi tiene dos UARTs:

- **UART0 (PL011):** UART hardware completo, estable y preciso
- **Mini UART:** UART simplificado, comparte reloj con GPU, menos confiable

Por defecto, el Bluetooth usa UART0, dejando solo el mini-UART para GPIO. Esto explicaba los problemas de timing.

Solución: Deshabilitamos completamente el Bluetooth para liberar UART0:

```
# En /boot/config.txt
dtoverlay=disable-bt
enable_uart=1
core_freq=250  # Fijar frecuencia core para UART estable

# Deshabilitar servicios
sudo systemctl disable hciuart
sudo systemctl disable bluetooth
```

Resultado: El puerto serial funcionó perfectamente después de esta configuración. La comunicación se volvió estable y predecible.

2.6.4. Problema 4: Cálculo Incorrecto de CRC

Descripción del problema: En nuestras primeras implementaciones del CRC, todas las respuestas del servo fallaban la verificación, incluso cuando visualmente (con analizador) los datos parecían correctos.

Análisis: El problema estaba en el orden de los bytes del CRC. Modbus usa formato little-endian para el CRC (byte bajo primero), pero inicialmente lo implementamos en big-endian.

Frame incorrecto (big-endian):

`[ID] [FC] [ADDR_H] [ADDR_L] [DATA_H] [DATA_L] [CRC_H] [CRC_L]`

Frame correcto (little-endian):

`[ID] [FC] [ADDR_H] [ADDR_L] [DATA_H] [DATA_L] [CRC_L] [CRC_H]`

Solución: Corregimos el orden al agregar el CRC:

```
crc = calculate_crc(command)
command.append(crc & 0xFF)           # CRC low byte primero
command.append((crc >> 8) & 0xFF)    # CRC high byte despues
```

Y en la verificación:

```
received_crc = data[-2] | (data[-1] << 8)  # Low byte primero
```

Después de esta corrección, el CRC funcionó perfectamente.

2.7. Comparación Práctica de Modos

2.7.1. Modo Simplex - Evaluación

Ventajas observadas:

- Implementación más simple: menos código y lógica
- Mayor throughput aparente (23.6 cmd/s)
- Sin overhead de cambio de dirección
- Útil para broadcast a múltiples dispositivos

Desventajas encontradas:

- Imposible verificar si comandos fueron ejecutados
- No se detectan errores de comunicación
- No se puede leer estado del dispositivo
- Inadecuado para aplicaciones críticas

Casos de uso identificados:

- Telemetría continua donde pérdidas ocasionales son tolerables
- Broadcast de setpoints a múltiples actuadores
- Sistemas donde otro mecanismo verifica la ejecución

2.7.2. Modo Half-Duplex - Evaluación

Ventajas observadas:

- Balance óptimo entre costo y funcionalidad
- Verificación completa con CRC de cada comando
- Solo requiere dos hilos (par trenzado)
- Protocolo Modbus RTU estándar facilita integración
- 98.2 % de confiabilidad demostrada

Desventajas encontradas:

- Overhead de 2-3ms por cambio de dirección
- Throughput moderado (14.5 cmd/s)
- Requiere gestión cuidadosa del control de dirección
- Un solo maestro puede transmitir a la vez

Conclusión personal: Este es el modo que recomendamos para la mayoría de aplicaciones industriales. La confiabilidad y capacidad de verificación justifican completamente el overhead adicional.

2.7.3. Modo Full-Duplex - Evaluación

Ventajas observadas:

- Mejor tasa de éxito (99.6 %)
- Latencia 46 % menor que half-duplex
- Sin overhead de turnaround
- Permite monitoreo continuo sin interrumpir comandos
- Mejor para control de alta velocidad

Desventajas encontradas:

- Requiere cuatro hilos (doble de cable)
- Costo hardware más alto (dos transceivers)
- Mayor complejidad de implementación
- Cableado más complicado

Conclusión personal: Solo se justifica en aplicaciones donde la latencia es crítica, como control de movimiento de muy alta precisión o sistemas con requisitos de tiempo real estrictos.

2.7.4. Para Configuración

1. Elegir baudrate conservador (9600-19200) para mayor confiabilidad
2. Deshabilitar Bluetooth en Raspberry Pi para liberar UART hardware
3. Fijar frecuencia del core para timing estable
4. Probar exhaustivamente antes de deployment
5. Documentar todas las direcciones Modbus de dispositivos

3. Referencias

1. PROFIBUS & PROFINET International (PI). (2024). *PROFIBUS Technology and Application - System Description*. Karlsruhe, Germany.
2. International Electrotechnical Commission. (2019). *IEC 61158: Industrial communication networks - Fieldbus specifications*. Geneva: IEC.
3. Institute of Electrical and Electronics Engineers. (2018). *IEEE 802.3 - Ethernet Standard*. New York: IEEE.
4. Telecommunications Industry Association. (1998). *TIA/EIA-485-A: Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems*. Arlington, VA.

5. Modbus Organization. (2012). *MODBUS Application Protocol Specification V1.1b3*. Hopkinton, MA.

6. Raspberry Pi Foundation. (2024). *Raspberry Pi Documentation - UART Configuration and GPIO Programming*. Cambridge, UK. Disponible en: https://www.raspberrypi.org/docum

7. Maxim Integrated Products. (2020). *MAX485/MAX487–MAX491: Low-Power, Slew-Rate-Limited RS-485/RS-422 Transceivers*. San Jose, CA: Maxim Datasheet.

8. Mackay, S., Wright, E., Reynders, D., & Park, J. (2004). *Practical Industrial Data Networks: Design, Installation and Troubleshooting*. Oxford: Newnes.

9. Zurawski, R. (2014). *Industrial Communication Technology Handbook, Second Edition*. Boca Raton: CRC Press.

10. Siemens AG. (2023). *SIMATIC NET - Industrial Ethernet Networks: Planning and Configuration Manual*. Nuremberg: Siemens Technical Documentation.

Anexo — Enlace al Repositorio

Repositorio GitHub:

https://github.com/JulianAlva24/TALLER_COMUNI.git

Este enlace se incluye como referencia del repositorio asociado al taller.

Fecha de inserción: 2025-10-24 04:35