Serial Transmission System: Implementation and Error Handling

Simphiwe Harvey Mankge

February 27, 2025

1 Introduction

This document details the implementation approach and error handling methodology for the serial communication system between a Raspberry Pi and a microcontroller. The system transmits structured data packets containing signal strength and Wi-Fi data using the UART protocol. The objective is to ensure reliable data transmission using robust techniques such as synchronization patterns, CRC verification, and error correction mechanisms.

2 Implementation Approach

The system consists of two main components:

- A **C-based sender** running on a microcontroller that transmits structured data packets over UART.
- A **Python-based receiver** on a Raspberry Pi that reads, parses, and verifies incoming data.

The following steps outline the implementation:

2.1 1. Defining the Data Structure

The transmitted data is encapsulated in a structured format:

Listing 1: Packet Data Structure

```c
typedef struct {
    uint8_t sync;         // Sync pattern for packet start
    uint8_t packet_type; // Packet type identifier
    int8_t rssi;          // Received Signal Strength Indicator
    uint16_t length;      // Length of the WiFi data
    int8_t wifi_data[MAX_WIFI_DATA]; // Dynamic array for WiFi CSI data
    uint32_t timestamp;  // Timestamp for logging
} SerialPacket;
```

2.2 2. Serial Communication Setup

On the microcontroller, serial communication is configured using UART functions. The low-level function `uart_write_bytes()` is used to transmit data:

Listing 2: Transmitting Data via UART

```
void send_serial_packet(SerialPacket *packet) {
    uint8_t buffer[PACKET_SIZE];
    serialize_packet(packet, buffer); // Convert struct to binary format
    uart_write_bytes((const char*)buffer, sizeof(buffer)); // Send via UART
}
```

On the Raspberry Pi, the Python script reads the serial data using the `pyserial` library:

Listing 3: Receiving Data in Python

```
import serial
ser = serial.Serial('/dev/tty006', 115200, timeout=1)
raw_data = ser.read(PACKET_SIZE)
```

2.3 3. Data Encoding and Packet Synchronization

Each packet includes a predefined **sync pattern** to identify the start of a new packet and prevent data misalignment.

Listing 4: Sync Pattern Definition

```
#define SYNC_PATTERN 0xA5
```

Before processing, the receiver scans for this pattern:

Listing 5: Finding the Sync Pattern

```
if raw_data[0] == 0xA5:
    print("Valid packet detected.")
```

2.4 4. Implementing Error Detection

We integrate a **Cyclic Redundancy Check (CRC)** to detect corrupted data. The sender computes a CRC before transmission:

Listing 6: Computing CRC Before Transmission

```
uint16_t compute_crc(uint8_t *data, size_t length) {
    uint16_t crc = 0xFFFF;
    for (size_t i = 0; i < length; i++) {
        crc ^= data[i];
    }
    return crc;
}
```

The receiver verifies the CRC:

Listing 7: Validating CRC in Python

```
received_crc = extract_crc(raw_data)
if received_crc == compute_crc(packet_data):
```

```
    print("Valid CRC. Data is intact.")
else:
    print("CRC error: Data may be corrupted.")
```

2.5 5. Handling Variable-Length WiFi Data

The system dynamically adjusts packet sizes based on the WiFi data length to avoid transmitting unused memory:

Listing 8: Handling Variable-Length Data

```
packet.length = rand() % MAX_WIFI_DATA;
```

2.6 6. Debugging and Testing

Since physical hardware is not required, a **virtual serial port** is used for testing:

Listing 9: Creating a Virtual Serial Port

```
socat -d -d pty,raw,echo=0 pty,raw,echo=0
```

The sender writes to `/dev/ttys005` while the receiver reads from `/dev/ttys006`.

3 Error Handling Methodology

3.1 1. Serial Port Connection Issues

If the serial port is unavailable or misconfigured, the receiver reports an error:

Listing 10: Handling Serial Port Errors

```
try:
    ser = serial.Serial('/dev/tty006', 115200, timeout=1)
except serial.SerialException as e:
    print(f"Error: {e}")
```

3.2 2. Handling Data Loss and Corruption

To mitigate data corruption, the following techniques are applied:

- Sync Pattern: Ensures packets are properly aligned.
- CRC Validation: Detects transmission errors.
- Timeout Handling: Avoids infinite waiting on serial reads.

3.3 3. Timeout Mechanism

If no data is received within a certain period, the system resets the connection:

Listing 11: Implementing Timeout

```
ser.timeout = 1
data = ser.read(PACKET_SIZE)
if not data:
    print("Timeout: No data received.")
```

3.4 4. Invalid Data Filtering

Invalid or incomplete packets are discarded:

Listing 12: Discarding Invalid Packets

```
if len(data) < PACKET_SIZE:
    print("Error: Incomplete packet received.")
    continue
```

3.5 5. Logging and Debugging

To facilitate debugging, error logs are stored:

Listing 13: Logging Errors

```
with open("error_log.txt", "a") as log_file:
    log_file.write(f"Error: Invalid packet at {time.ctime()}\n")
```

4 Conclusion

This document outlines the robust implementation approach and error handling techniques for a reliable serial communication system. The combination of structured packet transmission, CRC validation, and error recovery mechanisms ensures high data integrity and efficient debugging.