# Robust Two-Byte CRC Serial Communication Protocol (ESP32 & PC)

In this answer, we design a **foolproof serial communication protocol** between an ESP32 microcontroller and a PC. The protocol uses a 2-byte CRC (16-bit CRC) for error checking to ensure reliable data transfer. We will provide **C++ code for the ESP32 (PlatformIO Arduino framework)** and **Python code for the PC** side, along with explanations of the protocol structure.

## Protocol Overview

We will implement a simple request-response protocol suitable for sending servo commands from the PC to the ESP32 and getting feedback (e.g., current angle) from the ESP32 back to the PC. The protocol is designed to be robust against communication errors using a CRC-16 checksum. Key aspects of the protocol:

* **Frame Format:** Each message frame has a fixed structure with a **header**, **payload**, and **CRC**. Using a distinct header helps identify the start of a valid frame and maintain alignment.
* **Header:** We use a 2-byte header 0xAA 0x55 at the start of every frame to mark the beginning of a message.
* **Payload:** Contains a **Command ID (1 byte)** followed by the **data payload (2 bytes)**. In our servo example:
* PC → ESP32: Command ID = 0x01 (for "Set Servo Angle"), and the data (2 bytes) represents the target servo angle (e.g., 0–180 degrees).
* ESP32 → PC: Command ID = 0x02 (for "Feedback Response"), and the data (2 bytes) represents the feedback value (e.g., current angle or sensor reading).
* **CRC16 (2 bytes):** A 16-bit CRC is computed over the **payload bytes only** (i.e., the Command ID and data bytes). The CRC is appended in **little-endian** order (LSB first, then MSB). The CRC allows the receiver to detect any corruption in the ID or data. We will use the standard CRC-16 (Modbus/IBM) polynomial 0xA001 with initial value 0xFFFF for our calculations on both sides.

**Frame Structure (7 bytes total):**

[ Header (0xAA55) | ID (1 byte) | Data (2 bytes) | CRC16 (2 bytes) ]

* Header: 2 bytes (0xAA followed by 0x55)
* ID: 1 byte (identifies the command or message type)
* Data: 2 bytes (little-endian, specific to the command)
* CRC16: 2 bytes (little-endian, CRC of the ID + Data bytes)

Using a fixed header and known length makes the communication **robust**: - The receiver can **synchronize** on the header bytes and know exactly how many bytes to expect. - If a frame is corrupted or misaligned, the receiver will eventually find the next correct header sequence and realign. - The CRC ensures that any frame with corrupted data/ID is detected and ignored (or handled appropriately) rather than acting on bad data.

## Communication Process

The communication between PC and ESP32 will proceed as follows:

1. **PC (Master) Sends Command:** The PC constructs a frame with the header, command ID, data, and CRC. For example, to set the servo angle to 90°, the PC sends ID 0x01 and data 0x005A (90 in decimal is 0x5A in hex, sent as two bytes LSB first).
2. **ESP32 Receives and Verifies:** The ESP32 continuously listens on the serial port. It looks for the header bytes (0xAA55). Once detected, it reads the following bytes (ID, data, CRC). It calculates its own CRC over the received ID and data and compares it with the received CRC:
3. If the CRC is **valid**, the ESP32 knows the frame is correct. It will then act on the command. In our example, if ID = 1, it will set the servo to the specified angle. After executing the command, it prepares a response frame (with ID 2 for feedback) containing the current angle (or an acknowledgment) and its own CRC, and sends this back to the PC.
4. If the CRC is **invalid**, the ESP32 will **ignore** the frame (no action taken). The code will reset and wait for a new valid header sequence, ensuring it doesn’t act on corrupted data. (Optionally, one could send an error response or increment an error counter, but for simplicity we just drop invalid frames.)
5. **PC Receives Response:** After sending a command, the PC waits for the ESP32’s response frame. It checks the response’s header and CRC in the same way:
6. If the CRC matches, the PC accepts the data (e.g., reads the feedback angle and prints or uses it).
7. If not, the PC can ignore the response. In case of no valid response (CRC error or timeout), the PC can retry sending the command or handle the error as needed.
8. **Continuous Operation:** This protocol can be used in a loop – the PC can send multiple commands over time, and the ESP32 will respond to each. Both sides use the header and CRC to maintain synchronization and data integrity.

By following this process, we achieve a **reliable two-way communication** where both the PC and ESP32 validate data with CRC before using it. Next, we’ll provide the implementation details in code for both the ESP32 and the PC.

## ESP32 Code (C++ for PlatformIO/Arduino)

Below is the C++ code for the ESP32 (using the Arduino framework in PlatformIO). This code will run on the ESP32 (WROOM ESP32S dev board) and handle incoming serial frames, control a servo, and send back feedback. It uses the Arduino Servo library to control the servo motor.

**Key points of the ESP32 code:** - Initializes serial communication (e.g., Serial.begin(115200)) and attaches the servo to a specified pin in setup(). - Uses a **state machine** in loop() to parse incoming bytes from Serial. It waits for the header 0xAA 0x55, then reads ID, data bytes, and CRC bytes in sequence. - Once a full frame is received, it computes the CRC16 of the ID and data and compares with the received CRC: - If valid, it processes the command (ID 1 sets the servo angle) and prepares a response. - If invalid, it ignores the frame. - The response frame is constructed with its own header, an ID (0x02 for feedback), 2-byte data (the feedback value), and a CRC16 of the response payload, then sent via Serial.write().

Make sure to adjust the **servo pin** (servoPin) to the pin you have your servo connected to, and ensure the servo is powered appropriately. The code assumes the servo angle is given in degrees (0–180 range). If your servo command uses a different scale or units, you can adjust accordingly.

#include <Arduino.h>  
#include <Servo.h>  
  
Servo servo;  
const int servoPin = 18; // GPIO pin where the servo signal wire is connected  
  
// Compute CRC-16 (Modbus/IBM polynomial 0xA001) over an array of bytes  
uint16\_t computeCRC16(const uint8\_t \*data, size\_t length) {  
 uint16\_t crc = 0xFFFF; // initialize CRC  
 for (size\_t i = 0; i < length; ++i) {  
 crc ^= data[i];  
 for (uint8\_t bit = 0; bit < 8; ++bit) {  
 if (crc & 1) {  
 crc = (crc >> 1) ^ 0xA001; // polynomial 0xA001 (reflected 0x8005)  
 } else {  
 crc = crc >> 1;  
 }  
 }  
 }  
 return crc;  
}  
  
void setup() {  
 Serial.begin(115200);  
 servo.attach(servoPin);   
 servo.write(90); // (Optional) move servo to a safe starting position, e.g., 90°  
}  
  
void loop() {  
 // Static state for the state machine (persists across loop iterations)  
 static enum { WAIT\_HEADER1, WAIT\_HEADER2, WAIT\_ID, WAIT\_DATA1, WAIT\_DATA2, WAIT\_CRC1, WAIT\_CRC2 } state = WAIT\_HEADER1;  
 static uint8\_t recvId;  
 static uint16\_t recvData;  
 static uint16\_t recvCRC;  
  
 // Read incoming bytes and parse frame  
 while (Serial.available() > 0) {  
 uint8\_t byte = Serial.read();  
 switch (state) {  
 case WAIT\_HEADER1:  
 if (byte == 0xAA) {  
 state = WAIT\_HEADER2; // first header byte correct, move to next  
 } else {  
 state = WAIT\_HEADER1; // still waiting for first header byte  
 }  
 break;  
 case WAIT\_HEADER2:  
 if (byte == 0x55) {  
 state = WAIT\_ID; // header complete  
 } else {  
 // If second byte isn't 0x55, reset to look for header again.  
 state = WAIT\_HEADER1;  
 }  
 break;  
 case WAIT\_ID:  
 recvId = byte;  
 state = WAIT\_DATA1;  
 break;  
 case WAIT\_DATA1:  
 // Low byte of data  
 recvData = byte;  
 state = WAIT\_DATA2;  
 break;  
 case WAIT\_DATA2:  
 // High byte of data (combine with low byte)  
 recvData |= ((uint16\_t)byte << 8);  
 state = WAIT\_CRC1;  
 break;  
 case WAIT\_CRC1:  
 // Low byte of CRC  
 recvCRC = byte;  
 state = WAIT\_CRC2;  
 break;  
 case WAIT\_CRC2:  
 // High byte of CRC received, complete the frame  
 recvCRC |= ((uint16\_t)byte << 8);  
 // Now we have a full frame: [ID + Data] and the received CRC.  
 // Compute CRC on the received ID and data bytes to verify integrity.  
 uint8\_t payload[3];  
 payload[0] = recvId;  
 payload[1] = (uint8\_t)(recvData & 0xFF);  
 payload[2] = (uint8\_t)((recvData >> 8) & 0xFF);  
 uint16\_t calcCRC = computeCRC16(payload, 3);  
  
 if (calcCRC == recvCRC) {  
 // \*\*Valid frame received\*\* – process the command based on ID.  
 if (recvId == 0x01) {  
 // Command ID 0x01: Set Servo Angle  
 uint16\_t angle = recvData;  
 if (angle > 180) angle = 180; // clamp angle to servo range  
 // (If your servo accepts values in degrees 0-180.   
 // For other units/scales, adjust this logic accordingly.)  
 servo.write(angle); // set servo to the target angle  
  
 // Prepare feedback (current angle).   
 // If you have a sensor to read actual angle, use it here (e.g., analogRead and map to degrees).  
 // Otherwise, we will just echo the commanded angle as feedback.  
 uint16\_t currentAngle = angle;   
  
 // Construct response frame (ID 0x02 for feedback response):  
 uint8\_t respId = 0x02;  
 uint8\_t respPayload[3];  
 respPayload[0] = respId;  
 respPayload[1] = (uint8\_t)(currentAngle & 0xFF);  
 respPayload[2] = (uint8\_t)((currentAngle >> 8) & 0xFF);  
 uint16\_t respCRC = computeCRC16(respPayload, 3);  
  
 // Send the response frame: header, payload, CRC  
 Serial.write(0xAA);  
 Serial.write(0x55);  
 Serial.write(respPayload[0]); // ID  
 Serial.write(respPayload[1]); // Data LSB  
 Serial.write(respPayload[2]); // Data MSB  
 Serial.write((uint8\_t)(respCRC & 0xFF)); // CRC LSB  
 Serial.write((uint8\_t)((respCRC >> 8) & 0xFF)); // CRC MSB  
 }  
 // (You can add more `if/else if` blocks here for other command IDs in the future.)  
 } else {  
 // \*\*Invalid CRC\*\* – Frame is corrupted, ignore this frame.  
 // (Optionally, you could Serial.print an error or implement an error response.)  
 }  
 // Reset state to look for the next frame  
 state = WAIT\_HEADER1;  
 break;  
 } // switch  
 } // while Serial.available  
}

**How the ESP32 code works:** The state machine reads bytes one by one: - It waits for 0xAA and 0x55 in sequence to confirm the header. - Then it collects the next bytes as recvId, recvData (2 bytes), and recvCRC (2 bytes). - Once a full message is assembled, it computes calcCRC from the received ID and data, and compares it to the received CRC: - If they match, it processes the command. In our case, for ID 0x01, it writes the servo to the specified angle and then prepares a response (ID 0x02) with the current angle. - If they do not match, it ignores the frame (no action taken for safety). - After handling the frame (valid or not), it resets the state to WAIT\_HEADER1 to be ready for the next message. - This approach handles partial frames (the state machine will wait until all bytes arrive) and misaligned data (if a byte is lost or an invalid header is encountered, it will resync at the next 0xAA byte).

**Note:** Ensure that the servo is properly connected to the ESP32 (signal to the specified servoPin, plus power and ground). The Servo.write(angle) function expects an angle in degrees (0–180) by default. If your servo control uses a different scale (for example, microseconds), you would adjust the code accordingly. For feedback, if you have a potentiometer or sensor that measures the servo’s actual angle, you can replace the feedback calculation with a sensor reading (e.g., using analogRead and mapping the value to degrees). In this example, we simply echo back the commanded angle as the feedback.

## PC Code (Python)

Now, on the PC side, we’ll use Python (with the PySerial library) to send commands and receive responses. The Python code will construct the frame with the proper format and CRC, send it over a serial port, then read back the response and validate its CRC.

**Key points of the Python code:** - Uses pyserial to open the serial connection to the COM port that the ESP32 is connected to. - Constructs a command frame (with header, ID, data, CRC). We use Python’s struct module to pack data into bytes and a custom compute\_crc16 function to calculate the CRC to match the ESP32’s. - Sends the frame with ser.write(). - Waits for a response frame (7 bytes) using ser.read(). We set a timeout on the serial port so the read will return if no data is received within that time. - Parses the response: checks the header bytes, extracts the ID, data, and CRC. Then computes CRC on the received ID+data and compares with the received CRC to validate the integrity. - If the response is valid and the ID matches (expecting ID 0x02 for feedback), it prints out the feedback value (which could be the current angle). - If the response is invalid (wrong header, wrong CRC, or timeout/no data), it prints an error or takes appropriate action (in a real application, you might retry or handle the error accordingly).

Before running the Python code, make sure to install PySerial (pip install pyserial) and update the serial\_port variable to the correct port name for your system (e.g., "COM3" on Windows, "/dev/ttyUSB0" or similar on Linux/macOS).

import serial  
import struct  
  
# COM port settings – change 'COM3' to your actual port (e.g., 'COM5', '/dev/ttyUSB0', '/dev/ttyACM0', etc.)  
serial\_port = 'COM3'  
baud\_rate = 115200  
  
# Open the serial connection to the ESP32  
ser = serial.Serial(serial\_port, baud\_rate, timeout=1) # 1 second timeout for reads  
  
# Function to compute CRC-16 (polynomial 0xA001, initial value 0xFFFF), matching the ESP32's computation  
def compute\_crc16(data\_bytes: bytes) -> int:  
 crc = 0xFFFF  
 for b in data\_bytes:  
 crc ^= b  
 for \_ in range(8):  
 if crc & 1:  
 crc = (crc >> 1) ^ 0xA001  
 else:  
 crc >>= 1  
 return crc & 0xFFFF  
  
# Example: send a command to set servo angle to 90 degrees  
angle = 90 # target servo angle  
cmd\_id = 0x01 # Command ID for "Set Servo Angle"  
  
# Pack the data (angle) into two bytes (little-endian)  
data\_bytes = struct.pack('<H', angle) # <H means 16-bit unsigned little-endian  
# Construct payload = ID + data  
payload = bytes([cmd\_id]) + data\_bytes  
# Compute CRC16 on the payload  
crc\_value = compute\_crc16(payload)  
crc\_bytes = struct.pack('<H', crc\_value) # 16-bit CRC in little-endian format  
  
# Construct the full frame: header + payload + CRC  
frame = b'\xAA\x55' + payload + crc\_bytes  
  
print(f"Sending frame: {frame.hex()}") # Print out frame in hex for debugging (optional)  
ser.write(frame) # Send the frame to the ESP32  
  
# Read the response frame (expecting 7 bytes: 2 header + 1 ID + 2 data + 2 CRC)  
response = ser.read(7)  
if len(response) == 7:  
 # Parse response  
 header1, header2, resp\_id = response[0], response[1], response[2]  
 data\_low, data\_high = response[3], response[4]  
 crc\_low, crc\_high = response[5], response[6]  
 # Check header  
 if header1 == 0xAA and header2 == 0x55:  
 resp\_data = data\_low | (data\_high << 8)  
 resp\_crc\_received = crc\_low | (crc\_high << 8)  
 # Compute CRC on the received ID+data to verify  
 calc\_resp\_crc = compute\_crc16(bytes([resp\_id, data\_low, data\_high]))  
 if calc\_resp\_crc == resp\_crc\_received:  
 # CRC is valid – process the response  
 if resp\_id == 0x02:  
 print(f"Received valid feedback. Value = {resp\_data}")  
 else:  
 print(f"Received response with ID {resp\_id}, but was not the expected ID 0x02.")  
 else:  
 print("ERROR: Response CRC invalid! Data may be corrupted.")  
 else:  
 print("ERROR: Invalid response header received!")  
else:  
 print("ERROR: No response (timeout) or incomplete response received.")  
  
# Close the serial port when done (in a real application, you might keep it open for continuous communication)  
ser.close()

**How the Python code works:** This script opens the serial port and then prepares a single command as an example (setting the servo to 90°). Key steps: - We build the command frame by packing the command ID and angle into bytes, then appending the CRC. The struct.pack('<H', angle) call converts the 16-bit angle value into two bytes in little-endian order. - We then send the frame over serial using ser.write(). The example prints the hex representation of the frame for debugging (e.g., it might print something like aa55 01 5a00 a03b, which corresponds to header AA55, ID 01, data 005A (90 decimal), and CRC 3BA0 – low byte A0, high byte 3B). - After sending, we attempt to read 7 bytes back from the serial. The timeout=1 ensures the read will not block indefinitely – if no data is received within 1 second, the read will return with whatever bytes have arrived (possibly zero bytes). - We verify the response: - Check that the first two bytes are the header 0xAA55. - Extract the response ID, data, and CRC. - Compute our own CRC on the received ID and data (compute\_crc16(bytes([resp\_id, data\_low, data\_high]))) and compare with the received CRC bytes. - If everything checks out and the resp\_id is 0x02 (the expected feedback response), we print the feedback value. - If the CRC doesn’t match or the header is wrong, we print an error message. (In practice, if a CRC error occurs or no response is received, you could implement a retry mechanism or other error handling as needed.) - Finally, we close the serial port. In a real application, you might keep the port open and loop the send/receive process as needed.

## Usage and Testing

To test this setup:

* **Flash the ESP32** with the above C++ code (using PlatformIO or the Arduino IDE). Ensure the serial baud rate in Serial.begin() matches the Python script’s baud rate.
* **Run the Python script** on your PC. Make sure to adjust the serial\_port to the correct port name where your ESP32 is connected. The script will send a servo angle command (you can change the angle variable to test different values) and then listen for a response.
* **Observe the output:** The Python script will print the sent frame and the received feedback. For example, if you set angle = 90, you should see the Python script print something like Received valid feedback. Value = 90 if everything is working. This indicates the ESP32 correctly received the command (CRC OK, moved the servo) and sent back a valid response which the PC verified with CRC.
* You can extend this test by sending multiple commands (e.g., wrap the send/read in a loop or function) and even implement interactive control or a higher-level protocol on top of these frames. Each time, the CRC will protect against line noise or data corruption.

**Error handling:** If you intentionally alter a byte in the frame or if data gets corrupted, the CRC check will fail. The ESP32 will ignore that command (no action, no response), and the Python side will likely time out waiting for a response, or if it receives something with a bad CRC, it will print an error. This demonstrates the foolproof nature of the communication – invalid data is detected and not acted upon. In such cases, the PC can retry sending the command if needed.

## Conclusion

With the above protocol and code, we have a robust two-way communication channel between the PC and ESP32 with a 2-byte CRC ensuring data integrity. The **ESP32 C++ code** handles parsing and validating incoming commands (setting the servo angle and responding with feedback), and the **Python code** on the PC side constructs commands and verifies the ESP32’s responses. This framework can be expanded with additional command IDs and data fields as needed (just remember to adjust the CRC calculation and frame parsing accordingly). By following this structured approach, you can trust that any data acted upon by the ESP32 or PC is correct, making the system much more reliable in the face of serial communication errors or noise. Happy coding!