# Bulletproof Binary Communication Protocol (UART) for Servo Control

## Use a Binary UART Communication Channel

Since you’re connecting a computer to an ESP (e.g. ESP32) via USB, using a **UART serial link** in binary mode is a solid choice. The USB connection will appear as a virtual COM port, allowing you to send raw binary data to the ESP. UART is simple, full-duplex, and widely supported – ideal for sending servo commands and sensor data quickly and reliably[[1]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Cars%20use%20UART%20for%20diagnostic,flexible%20enough%20for%20various%20applications)[[2]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Manufacturing%20teams%20use%20UART%27s%20detailed,check%20product%20quality%20before%20shipping). Make sure both sides use the same **baud rate** (e.g. 115200 bps or higher if needed) and frame format (typically 8 data bits, no parity, 1 stop bit, i.e. **8N1**) for proper communication. UART’s ease of use and flexibility make it well-suited for robust binary protocols in embedded applications[[1]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Cars%20use%20UART%20for%20diagnostic,flexible%20enough%20for%20various%20applications).

**Why not use PWM or other methods?** The servo itself will still be driven by the ESP’s PWM output (standard 50 Hz servo pulses). The binary UART link is only for communications *between the PC and ESP*, not to replace the servo’s PWM signal. By sending high-level **position commands in binary** to the ESP, you avoid the jitter and limitations of analog PWM over a long cable. Alternatives like I²C or SPI aren’t practical here because the PC’s interface to the ESP is via USB (serial). Thus, UART over USB is the appropriate transport.

## Define a Structured Binary Message Frame

To make the link **“bulletproof,” design a clear packet/frame structure** for your data. This ensures that both ends can parse commands and data unambiguously from the raw byte stream. A typical robust frame format for UART communication is:

* **Header** – 1–2 fixed bytes to mark the start of a packet (also called **sync bytes** or **start-of-frame delimiters**). For example, you might choose 0xAB 0xCD as a 2-byte header sequence[[3]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=frame%20has%20unique%20identifiers%20such,that%20authenticate%20the%20communicating%20device). These unique bytes act as a signature to help the receiver identify the beginning of a valid message[[4]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Custom%20frame%20protocols%20add%20vital,that%20authenticate%20the%20communicating%20device).
* **Length** – 1 byte indicating the number of bytes in the payload (or in the entire packet, depending on design). This lets the receiver know how many bytes to read for the message[[5]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=%2A%201,16).
* **Command/ID** – 1 byte to specify the **message type or command**. This could define whether the packet is a servo position command, a sensor data report, an acknowledgment, etc. (You can also treat this as part of the payload, but dedicating a field for it adds clarity.)[[6]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=The%20protocol%20structure%20has%20command,accidental%20data%20alterations%20during%20transmission)
* **Payload Data** – N bytes of binary data containing the actual information (e.g. the servo angle or sensor readings). The length byte should equal N (or N plus the size of the command byte, depending on definition). Make sure to use a consistent data format for these values (see **Binary Data Encoding** below).
* **CRC** – 1–2 bytes containing a **cyclic redundancy check** value computed over the packet’s content (usually covering the length, command, and payload fields)[[7]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=It%20doesn%27t%20make%20sense%20to,to%20be%20designed%20like%20this)[[8]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=2). A 16-bit CRC (two bytes) is commonly used for strong error detection, although a CRC-8 (one byte) could be sufficient for very small payloads[[9]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=,16). A CRC placed at the end of the frame lets the receiver verify the integrity of the entire message.

*Example of a custom binary UART frame structure with header, command, length, data, and CRC fields*[*[3]*](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=frame%20has%20unique%20identifiers%20such,that%20authenticate%20the%20communicating%20device)*.*

By using a fixed header and length field, the receiver can determine where each packet starts and ends. **Normal UART protocols follow this pattern**: for instance, “1–2 sync bytes, 1 length byte, N data bytes, and a 2-byte CRC-16” is a well-proven design[[7]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=It%20doesn%27t%20make%20sense%20to,to%20be%20designed%20like%20this). The header bytes (e.g. 0xABCD) serve as a reliable marker of the frame start, and the length tells how far to read until the packet is complete (where the CRC is located). This structured approach prevents the data stream from being misinterpreted.

## Binary Data Encoding and Commands

When encoding values (like servo angles or sensor readings) in the payload, use fixed-size binary representations and **consistent endianness** on both sides. For example:

* If your servo angle is an integer from 0–180 degrees, you could send it as a single byte (0x00–0xB4 in hex). For finer resolution (say 0–1000 or a specific pulse width in microseconds), use 2 bytes (e.g. a 16-bit integer).
* For sensor data, choose an appropriate binary format (int16, float32, etc.) depending on the range and precision needed. If sending multiple values (e.g. X, Y, Z from an IMU), decide the order and size of each field in the payload.

**Be explicit about data formats:** Both the PC and ESP code should agree on the exact data types. For instance, if you choose a 16-bit little-endian integer for the angle, ensure you pack and unpack those two bytes consistently in that order. Never rely on language-specific defaults like an int without confirming its size – an int on one system might be 32 bits while on another it could be 16 bits[[10]](https://forum.arduino.cc/t/binary-serial-communication-esp32-to-arduino-uno/1258692#:~:text=using%20native%20int,the%20Uno%20it%27s%2016%20bit). It’s safest to use fixed-width types (e.g. uint16\_t in C, or specifying byte order when building the packet in Python) so that the binary data aligns exactly between the computer and the ESP.

Using binary encoding means you send the data in its raw numeric form, rather than as ASCII text. This makes communication **faster and more efficient** (e.g. one byte can represent 255, whereas sending the string "255" would take three bytes plus a delimiter). It also avoids any need to parse strings on the microcontroller. With your protocol defined, the ESP code will simply read bytes from UART and interpret them according to the frame structure (header → length → command → data → CRC).

## Error Detection with CRC

Including a **CRC checksum** in each packet is key to a “bulletproof” communication. The CRC allows the receiver to detect if any bits were corrupted in transit (due to noise or other issues) before acting on the data[[6]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=The%20protocol%20structure%20has%20command,accidental%20data%20alterations%20during%20transmission). A 16-bit CRC is recommended for robustness – it will **detect virtually all single-bit and double-bit errors, and even large burst errors with extremely high probability**[[11]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=A%20good%20generator%20polynomial%20will,of%20all%20errors). In fact, a well-chosen CRC polynomial will catch *all* single- and double-bit flips, any odd number of bit errors, all burst errors up to the CRC’s bit-length, and over 99.9% of longer bursts[[11]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=A%20good%20generator%20polynomial%20will,of%20all%20errors). This level of error checking is far superior to the UART’s basic parity bit (which can only detect a single-bit error in a one-byte frame).

**How to use the CRC:** At the sender (PC side for commands or ESP side for sensor data), compute the CRC over the message bytes *before* the CRC field (typically covering the length, command, and payload). Then append the CRC bytes to the packet. On the receiving end, once the full packet is received, recompute the CRC over those same fields and compare it to the received CRC value[[12]](https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html#:~:text=,Formula). If they match, the data is presumed valid; if not, you know the packet was corrupted and should be discarded.

*Example:* If your packet’s bytes (excluding the header and CRC) are [Length][Command][Data...], you would run the CRC algorithm over this sequence of bytes. The result (say a 16-bit number) gets sent as the final two bytes of the packet. The receiver performs the same CRC calculation on the incoming bytes and verifies the result matches[[13]](https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html#:~:text=The%20cycling%20redundancy%20checking%20formula,computations%20on%20the%20receiver%E2%80%99s%20end).

Using a CRC satisfies the “bulletproof” requirement for error **detection**. Since you indicated CRC alone is fine (no need for full error *correction* or encryption), this will allow the system to detect errors and then handle them appropriately (see next section). CRCs are used in many reliable protocols (for example, Modbus RTU uses a 16-bit CRC at the end of each frame) because they are fast to compute and very effective at catching transmission errors.

## Robustness and Error Handling Strategies

Beyond just detecting errors, a truly bulletproof communication scheme should gracefully handle them and maintain synchronization:

* **Resynchronization on Errors:** Design the receiver logic as a simple state machine that continuously looks for the header bytes. If a packet’s CRC check fails, the receiver should **discard that packet** and then listen for the next header sequence in the stream rather than processing bad data. The unique header bytes make it unlikely that a random error will mimic a valid header; even if it does, the CRC is almost certain to fail and be caught. By scanning for the known header after a failure, the system will **resynchronize to the next valid packet boundary** and not get stuck in a misaligned state[[14]](https://www.embeddedrelated.com/showarticle/113.php#:~:text=You%20can%20still%20have%20errors,and%20you%20change%20the%20meaning). (If you use two header bytes, you might implement this by shifting one byte at a time through the stream looking for the first header byte, then checking if the next byte matches the second header, etc.)
* **Byte-Stuffing (Escaping):** If there’s a chance that your binary data could naturally contain the **header byte pattern** or other reserved values, you should use an escape mechanism. For instance, choose a special escape byte (like 0x7D) and define a rule that if that byte or a header byte occurs in the payload, it gets replaced with an escape sequence (and unescaped on the receiving end)[[15]](https://www.embeddedrelated.com/showarticle/113.php#:~:text=Another%20option%20is%20to%20use,what%27s%20next%20as%20raw%20bytes). This ensures that the header and any end-of-frame markers never accidentally appear in the middle of a packet and confuse the framing. An alternative framing method is **COBS (Consistent Overhead Byte Stuffing)**, which transforms data so that a zero byte can be used as a guaranteed packet delimiter[[16]](https://www.embeddedrelated.com/showarticle/113.php#:~:text=scheme%2C%20when%20I%20ran%20across,should%20use%20COBS%2C%20but%20you)[[14]](https://www.embeddedrelated.com/showarticle/113.php#:~:text=You%20can%20still%20have%20errors,and%20you%20change%20the%20meaning). However, if your header is sufficiently unique (e.g. two uncommon bytes in a row) and you include a CRC and length, the likelihood of false framing is extremely low. Many simple protocols forego complex stuffing and rely on the combination of unique headers + length + CRC to maintain alignment.
* **Acknowledgments & Retries:** Consider whether you need an ACK/NACK scheme. With a CRC, the receiver can detect bad packets but what then? In a one-way telemetry scenario, you might simply drop the bad packet. But in your case, where the PC sends commands that the ESP *must act on*, you might want the ESP to send back an acknowledgment message for critical commands. For example, when the ESP receives a servo position command and it passes CRC, it could respond with a short “OK” packet (or even just echo the command with a flag) to confirm it was received intact. If the PC doesn’t get an ACK in time, it can resend the command. This handshake ensures **no lost command** goes unnoticed[[17]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=With%20CRC%20,transmit%20a%20dropped%20message). On the other hand, if you are streaming commands or data continuously (e.g. updating servo position 50 times per second), you might design the system to tolerate occasional drops (the next update will arrive shortly). Choose a strategy appropriate for your application’s real-time needs. “Bulletproof” in this context means **fault-tolerant** – either the system recovers from errors by retrying, or it safely ignores a bad packet and continues with the next.
* **Low Latency Considerations:** The binary protocol as described is very lightweight – just a few bytes of overhead (header, length, CRC) – so it inherently has low latency. To keep it snappy, avoid overly large packets that could introduce delays. If you have a lot of sensor data to send, consider sending it in smaller chunks or at a reasonable update rate. UART at high baud (e.g. 1,000,000 bps) can easily handle many kilobytes per second, but if you approach the limit, ensure you’re not overflowing any buffers. The ESP’s UART has FIFO buffers (and USB CDC drivers have buffers too) that can handle bursts of data, but if you plan on **very high throughput**, you might enable hardware flow control (RTS/CTS)[[18]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Flow%20control%3A%20hardware%20vs%20software,methods)[[19]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Flow%20control%20prevents%20data%20loss,is%20asserted%20with%20CTS%20enabled) or insert small delays to let buffers clear. For typical command-and-feedback usage, this won’t be an issue.
* **Security (if needed):** You mentioned encryption isn’t required, so “bulletproof” is focused on reliability rather than security. If in the future you did need to secure the link (to prevent tampering or eavesdropping), you would have to encrypt the payload. This can be done by applying AES or similar encryption to the data bytes, but note that you’d then likely apply CRC *after* decryption on the far end (since encrypted data would make the CRC look like random). Encryption isn’t trivial over UART and will add processing time, so only add it if necessary. For now, a CRC-guarded binary protocol will be both fast and robust against errors, which covers most reliability needs.

## Conclusion

By using a binary protocol over UART with a well-defined frame structure, you can achieve **reliable, “bulletproof” communication** between your computer and the ESP controlling the servo. In summary, the approach is: **frame your data with unique header bytes, include a length and message type, encode the values in binary form, and append a CRC for error detection**. This design – similar to those used in industrial and robotics protocols – ensures that any corruption is caught and that messages aren’t misinterpreted[[3]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=frame%20has%20unique%20identifiers%20such,that%20authenticate%20the%20communicating%20device)[[7]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=It%20doesn%27t%20make%20sense%20to,to%20be%20designed%20like%20this). With proper handling of the CRC (and retries or acknowledgments if needed), the system will be robust against noise or data loss.

Implementing this will involve writing matching send/receive code on both the PC and the ESP. On the PC side, you can use a serial library (e.g. Python’s pySerial or C++ serial ports) to format bytes according to the protocol. On the ESP side, use the UART API to read bytes and assemble packets (e.g. in Arduino, read bytes in loop and parse when header/length criteria are met). Test the communication thoroughly – send a variety of commands and ensure the ESP correctly parses them and that sensor data is correctly reconstructed on the PC. With the structured binary protocol and CRC checks in place, you’ll have a communication link that is fast, efficient, and resilient to errors[[6]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=The%20protocol%20structure%20has%20command,accidental%20data%20alterations%20during%20transmission)[[13]](https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html#:~:text=The%20cycling%20redundancy%20checking%20formula,computations%20on%20the%20receiver%E2%80%99s%20end). This way, you can confidently send servo positions and receive sensor feedback without worrying about miscommunication or data glitches. Good luck with your project!

**Sources:** Custom UART frame design principles[[3]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=frame%20has%20unique%20identifiers%20such,that%20authenticate%20the%20communicating%20device)[[7]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=It%20doesn%27t%20make%20sense%20to,to%20be%20designed%20like%20this); CRC error-detection capabilities and usage[[11]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=A%20good%20generator%20polynomial%20will,of%20all%20errors)[[17]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=With%20CRC%20,transmit%20a%20dropped%20message); Arduino forum example on data size consistency[[10]](https://forum.arduino.cc/t/binary-serial-communication-esp32-to-arduino-uno/1258692#:~:text=using%20native%20int,the%20Uno%20it%27s%2016%20bit); Analog Devices – UART framing with headers/trailers and CRC[[4]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Custom%20frame%20protocols%20add%20vital,that%20authenticate%20the%20communicating%20device)[[12]](https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html#:~:text=,Formula).

[[1]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Cars%20use%20UART%20for%20diagnostic,flexible%20enough%20for%20various%20applications) [[2]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Manufacturing%20teams%20use%20UART%27s%20detailed,check%20product%20quality%20before%20shipping) [[3]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=frame%20has%20unique%20identifiers%20such,that%20authenticate%20the%20communicating%20device) [[4]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Custom%20frame%20protocols%20add%20vital,that%20authenticate%20the%20communicating%20device) [[6]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=The%20protocol%20structure%20has%20command,accidental%20data%20alterations%20during%20transmission) [[18]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Flow%20control%3A%20hardware%20vs%20software,methods) [[19]](https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS#:~:text=Flow%20control%20prevents%20data%20loss,is%20asserted%20with%20CTS%20enabled) UART Protocol Explained: From Basics to Advanced Implementation - Evelta Electronics

<https://evelta.com/blog/uart-protocol-explained-from-basics-to-advanced-implementation/?srsltid=AfmBOopPckF3tZwpTqmWNWBNKa4DAdcuKAg4YBoXLcz2mipBuvlMNTTS>

[[5]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=%2A%201,16) [[7]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=It%20doesn%27t%20make%20sense%20to,to%20be%20designed%20like%20this) [[8]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=2) [[9]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=,16) [[11]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=A%20good%20generator%20polynomial%20will,of%20all%20errors) [[17]](https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation#:~:text=With%20CRC%20,transmit%20a%20dropped%20message) serial - Ways to send data with CRC validation - Electrical Engineering Stack Exchange

<https://electronics.stackexchange.com/questions/597248/ways-to-send-data-with-crc-validation>

[[10]](https://forum.arduino.cc/t/binary-serial-communication-esp32-to-arduino-uno/1258692#:~:text=using%20native%20int,the%20Uno%20it%27s%2016%20bit) Binary serial communication ESP32 to Arduino Uno - Programming - Arduino Forum

<https://forum.arduino.cc/t/binary-serial-communication-esp32-to-arduino-uno/1258692>

[[12]](https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html#:~:text=,Formula) [[13]](https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html#:~:text=The%20cycling%20redundancy%20checking%20formula,computations%20on%20the%20receiver%E2%80%99s%20end) UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter | Analog Devices

<https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html>

[[14]](https://www.embeddedrelated.com/showarticle/113.php#:~:text=You%20can%20still%20have%20errors,and%20you%20change%20the%20meaning) [[15]](https://www.embeddedrelated.com/showarticle/113.php#:~:text=Another%20option%20is%20to%20use,what%27s%20next%20as%20raw%20bytes) [[16]](https://www.embeddedrelated.com/showarticle/113.php#:~:text=scheme%2C%20when%20I%20ran%20across,should%20use%20COBS%2C%20but%20you) Help, My Serial Data Has Been Framed: How To Handle Packets When All You Have Are Streams - Jason Sachs

<https://www.embeddedrelated.com/showarticle/113.php>