The challenge is that serial data arrives as a stream not in complete packets. The “readChannel” function can be called at any time in the middle of a data transmission when no data is available.

If we try to read and parse the data immediately on each function call we will get garbage. For example:

If you call “readChannel” and the stream only has "<1200 1500 ", parsing right then would fail because you don't have all 8 channels. The next call might get "1300 1400>" which is also incomplete on its own.

Another thing is that we already had a controller communication system that prints `<1500 1500 1500 1500 ...>` packets from a CRSF receiver.
But after switching to a new protocol we only need to translate or reformat the incoming serial data not rebuild the entire control code.

The new controller still sends channel values as text (like `<1200 1300 1400 ...>`) but the data may arrive incomplete.

So we must accumulate incoming data into a buffer until we can be certain that we have a complete and valid packet.Only then we’ll be able to parse it and make the data available to the rest of the program.

SOLUTION:

Instead of a one-shot function we need a small system with memory state.

So we can use a global variable:

```
#define NUM_CHANNELS 8

int channels[NUM_CHANNELS];        // The final, validated channel values
bool newDataAvailable = false;     // A flag to signal new data is ready
String serialBuffer = "";          // The accumulator for incoming serial
data
```

Here:

- **Channels[ ]:** It's the final output, the array of 8 integers that the rest of the old code will use.
- **newDataAvailable:** This flag acts like a "data ready" light. The loop() can check this flag and only use the new channels values when it knows that they have been freshly updated from a complete packet.

- **serialBuffer:** It's a string that accumulates characters over multiple calls to readChannel. It allows us to collect a fragment like "<1200 1500 " and then later append "1300 1400>" to form a complete string "<1200 1500 1300 1400>" that we can then process.

FOR THE “readChannel()” function:

We can convert this function into a state machine data collector. So what I can do is here I can write:

```
) {
void readChannel() {
    while (Serial1.available() > 0) {
        char incomingChar = Serial1.read();

        if (incomingChar == '<') {
            serialBuffer = "";
        }
        else if (incomingChar == '>') {
            processPacket(serialBuffer);
            serialBuffer = "";
        }
        else {
            serialBuffer += incomingChar;
        }
    }
}
```

Here:

while (Serial1.available() > 0): We don't just read one character per loop. We read every single character that is currently available in the serial buffer. This ensures that we process data as fast as it comes in and don't let the serial buffer overflow.

if (incomingChar == '<'):
The ‘<’ character is our packet start marker. When we see it we know a new packet is beginning. Any old data in the serialBuffer is discarded because it was either already processed or incomplete. This ensures we always start fresh with a new packet.

else if (incomingChar == '>'):
The '>' character is our packet end marker. This is the trigger that tells us we might have a complete packet. When we see it, we call **processPacket(serialBuffer)** with the accumulated contents and then clear the buffer and it will be ready for the next packet.

else {
serialBuffer += incomingChar;
}:
For all other characters (digits, spaces) we simply append them to our accumulating serialBuffer. This is how we build the string "1200 1500 1300 1400" from the stream.

Also, Serial communication doesn't guarantee that a whole message arrives at once sometimes you get just part of it. This line-by-line reading ensures you collect everything piece by piece until a full message (`<...>`) is complete. That's why we used conditional statements to balance it out so that it extracts only the datas.

For the loop function :

```
void loop() {
    readChannel();
    if (newDataAvailable) {
        newDataAvailable = false;
        executeWheelControl();
        printChannels();
    }
    updateStatusLEDs();
    readSensors();
}
```

This will call '**`readChannel()`**' continuously to keep updating the `channels[]` array and print it for debugging. This will simulate how the control loop will read the latest valid channel data in real time just like the rover wheels or arm would.