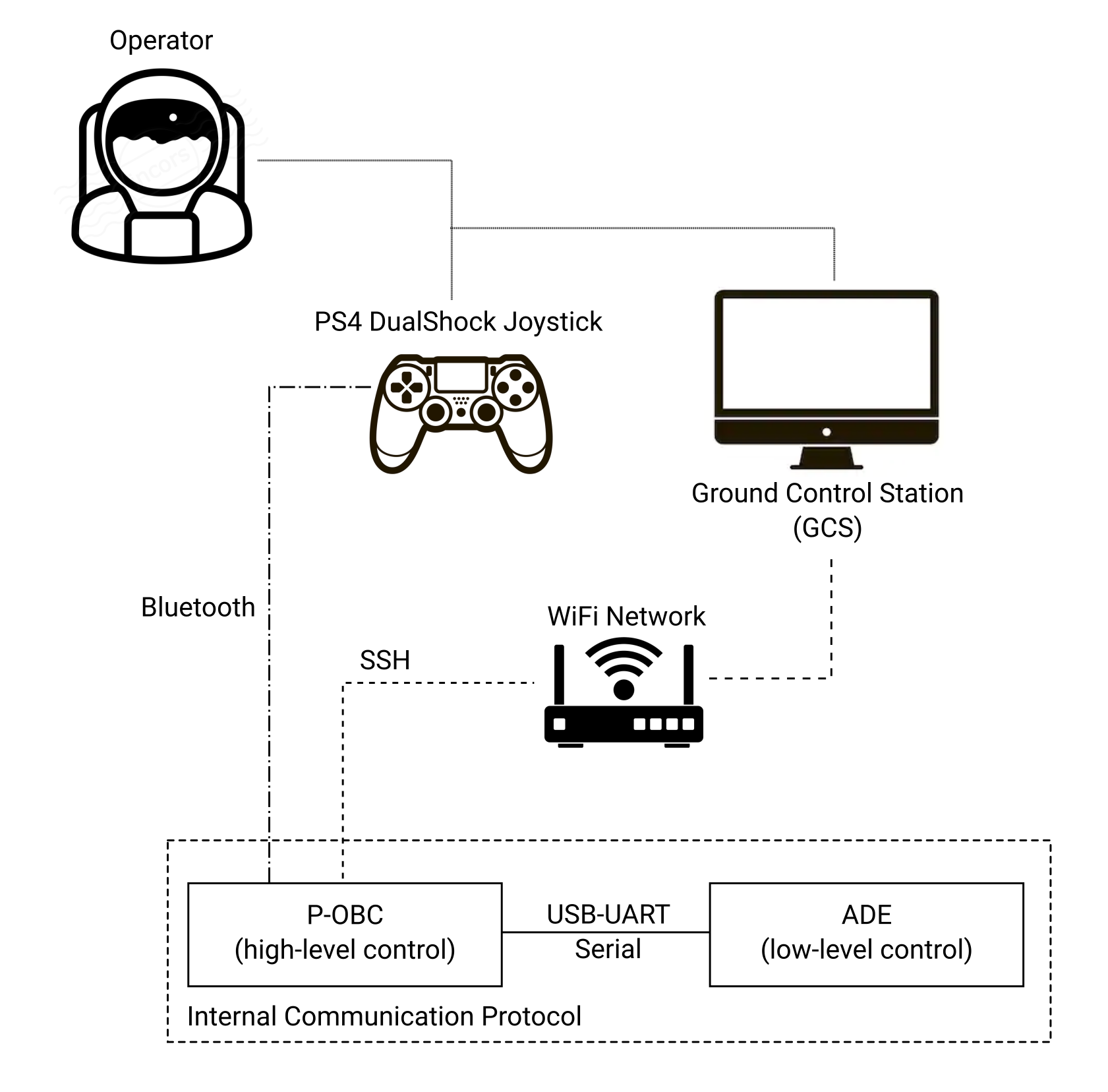
# Communication system.

## Overview.

EX1 communication system architecture was designed, for the most part, with the objective to maximize reliability. Common errors associated with data transmission include the reception of erroneous or uncompleted messages, the presence of noise in the data, interferences, missing incoming message characters, and synchronization errors, among others. Given the importance associated with the transmission and reception of data, a First In First Out, or FIFO, paquet structure was implemented as part of EX1 internal communication protocol (details in Section 7.4). Specific routines for the identification, evaluation, manipulation, and elimination of such errors were implemented. Overall, three main lines of communication can be identified as presented in Fig. 7-1.



***Fig. 7-1.*** *Simplified communication system architecture.*

One should note that, despite the pursuit of reliability in the communication system, certain components of the current architecture were chosen due to their immediate availability and ease of use. One such example is the teleoperation joystick. While bluetooth provides a direct and easy connection to the P-OCB, its short range, low data security, and low speed makes it a poor candidate as a long-term solution. Future users of the platform are encouraged to further investigate other alternatives such as the addition of a radio control module.

## Internal communication protocol.

EX1 internal communication protocol between the P-OBC (high-level control tasks) and the micro-controllers of the ADE (low-level control tasks) was implemented through **USB-UART serial communication** by which **custom packets of information** were exchanged. These packets contained, when sent by the P-OBC, information about the commands triggered through the joystick by the operator and, when sent by the micro-controllers, telemetry data fed back to the GCS by the proprioceptive sensors of the ADE (i.e., motor encoders, current sensors, rocker potentiometers) and the IMU readings (accelerations, orientation, and internal temperature).

As specified in Section 4.3., FTDI modules are being used to handle the conversion from UART-TTL to USB and vice versa. For this reason, in order for the ADE to communicate with the P-OBC, all it is required is to send our custom framed packets under UART protocol (i.e., one byte at a time). In the opposite direction—from the P-OBC to the micro-controllers of the ADE— command packets were defined and framed in the same way but the final transmission via USB was handled by a specific ROS serial library.

### UART driver.

In order for messages to be communicated and decoded properly by the P-OBC, the UART controller of the moto controllers were set with the following parameters.

***Table 7-1****. UART communication settings.*

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Value | Units | Definition |
| Baud rate | 115200 | bit/s | Frequency of transmission |
| Stop bit(s) | 1 | bit | Number of stop bit(s). Can be 1, 1.5, or 2. |
| Data size | 8 | bit | Number of useful (non-framed) bits in a frame. Can be 8 or 9. |
| Parity | N.A. | - | Presence or not of a parity bit. Can be 0 or 1 bit, even or odd. |

These settings imply that the efficiency of transmission is 80%[[1]](#footnote-0). In order words, for each byte we want to send, we need to send 10 bits. The maximum sending frequency is then 92,169 bits/s[[2]](#footnote-1). Each sending device must be ensured not to exceed this maximum rate.

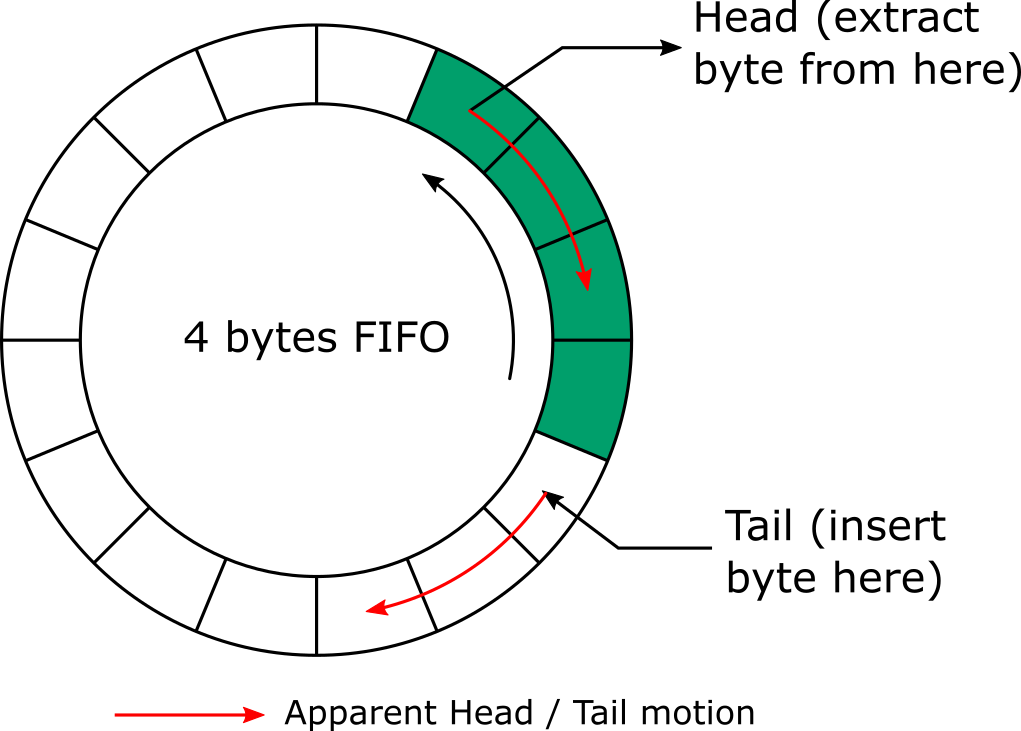
Once the settings are defined, reading and sending data through UART is rather simple. One major drawback of the UART protocol is, however, that only two registers exist on which devices can read and write raw bytes of information. When a new byte is received, this is automatically written in the register. The reader can now access this register at any time to retrieve the byte sent. But the register can only store one byte at a time. This means that if before the initial byte sent is retrieved and read, a second byte is sent, this second byte will be automatically discarded from the register precluding the possibility to read it. In addition, a full message is often formed by more than one byte. A process to retrieve, read, and temporarily store these bytes of information until a full message is received had to be implemented: this is what is called a **buffer**.

*Note: One could set up a UART communication without the need of a buffer by forcing the writer to wait before sending a byte until the previously sent byte has been properly read. This, however, introduces a lot of inefficiencies to the communication protocol.*

### Ring FIFO Buffer.

A **ring First In First Out (FIFO) or queue buffer** was implemented in the internal communication protocol. This makes storing bytes of information (more than one at a time) possible until they are fetched and processed by the reader. The FIFO or queue structure ensures that, regardless of the number of bytes sent, the information is always stored and processed in the correct order; i.e., from when the last processing ended.

FIFO structures make use of a ring. A ring consists of a circular structure that avoids the need for shifting elements in the queue from tail to head every time a new byte is received and stored. Instead, the queue advances around the ring (looping the circular structure). The head and the tail of the queue simply moves backwards (the tail grows) around the ring as new information arrives. An example of how a FIFO ring works is presented in Fig. 7-2.



***Fig. 7-2.*** *How a FIFO ring works.*

When utilizing FIFO ring structures, two FIFOs are actually being used: a transmit FIFO and a receive FIFO. Their functionality is slightly different. If interested, more information can be read online about FIFO rings and their functionality.

### Packet architecture.

A packet can be defined as a complete message that contains direct information—useful information for the reader, e.g., a command and its value—and safety information—information used to verify the integrity of the message being read. A packet is composed of different sets of information or arguments (see *Arg. Name* column in Table 7-2). The size of any packet will be between 8 and 24 bytes. The structure of the packets used for EX1 internal communication protocol is described in Table 7-2.

***Table 7-2****. EX1 internal communication protocol packet description.*

|  |  |  |
| --- | --- | --- |
| Arg. Name | Size [bytes] | Description |
| ID | 2 | A unique message identifier. |
| CMD | 1 | Specific command code to be executed. |
| LEN | 1 | Size (in bytes) of the DATA section. Max value is 16. |
| DATA\* | LEN | Arguments associated with the command CMD. |
| CRC | 4 | CRC code to ensure the message is free or errors. |

\*DATA is an optional argument used when LEN ≠ 0.

In addition to a custom packet definition, a **safety verification procedure** was implemented in order to make sure that packets being read and ultimately processed are free of errors. A packet would be qualify as uncorrupt (i.e., good to be processed by the reader) if:

* CMD corresponds to a valid command within the dictionary (see Section 7.2.4.).
* LEN should match the required data length of CMD.
* The total size of the packet should always be equal to 8 bytes + LEN.
* The CRC should correspond to the packet being read (more details on the CRC process implemented are presented in Section 7.2.5.)

If any of the previously described requirements is not met, the packet will be automatically discarded. Uncorrupt packets will be processed by the reader (the P-OBC or the micro controllers depending on what information is being shared).

### Dictionary of commands.

Each packet transmitted corresponds to either information requested by the P-OBC (e.g., telemetry data) or commands sent by the P-OBC (e.g., move forward at 0.2 m/s). Table 7-3 describes the different commands and Table 7-4 describes the different pieces of information available within the internal communication protocol.

***Table 7-3****. Commands sent by the P-OBC.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Command name | Description | CMD [hex.] | LEN | DATA (unit) [Type] |
| LIGHTS\_ON | Turn on a built-in debug LED | 0x01 | 0 | N.a. |
| LIGHTS\_OFF | Turn off a built-in debug LED | 0x02 | 0 | N.a |
| SETSPEED\_FRONT | Set the speed of the FDM | 0x03 | 4 | Speed (deg/s) [float] |
| SETSPEED\_REAR | Set the speed of the RDM | 0x04 | 4 | Speed (deg/s) [float] |
| SETSTEER\_FRONT | Set the angle of FSM | 0x05 | 4 | Angle (deg) [float] |
| SETSTEER\_REAR | Set the angle of the RSM | 0x06 | 4 | Angle (deg) [float] |

FMD= Front driving motors, RDM= Rear driving motors, FSM = Front steering motors, RSM = Rear steering motors.

***Table 7-4****. Data sent to the P-OBC.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data name | Description | CMD [hex.] | LEN | DATA (unit) [Type] |
| VAL\_SPEED\_FRONT | FDM speed | 0x07 | 4 | Speed (deg/s) [float] |
| VAL\_SPEED\_REAR | RDM speed | 0x08 | 4 | Speed (deg/s) [float] |
| VAL\_ANG\_FRONT | FSM angle | 0x09 | 4 | Angle (deg) [float] |
| VAL\_ANG\_REAR | RSM angle | 0x0A | 4 | Angle (deg) [float] |
| VAL\_CURR\_FRONT | Current through the FDM | 0x0B | 2 | ADC\_out (/)\* [uint16] |
| VAL\_CURR\_REAR | Current through the RDM | 0x0C | 2 | ADC\_out (/) [uint16] |
| VAL\_ANG\_RCK | Rocker angle | 0X0D | 4 | Angle (deg) [float] |

FMD= Front driving motors, RDM= Rear driving motors, FSM = Front steering motors, RSM = Rear steering motors.

\*The ADC is a 12-bit ADC. This means that the resolution of the ADC is 4096 (2^12) discrete levels. 5V = 4096.

### Cyclic Redundancy Check (CRC).

*“A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short check value or CRC value attached, based on the remainder of a polynomial division of their contents.”*

In our case, the CRC consists of a 32 bits value calculated based on the remaining arguments of the transmitted packet. The CRC of each packet will be first calculated by the writer (the transmitter) and attached to the remainder of the packet. The reader (receiver) will then receive the packet, retrieve all the arguments except for the CRC, recalculate the CRC, and compare it to the CRC attached to the packet by the writer. The value provided by the use of CRC is that if during transmission the information contained in the packet or the CRC itself is altered, upon recalculating the CRC on the reader side this type of data corruptions can be easily detected (CRC on the reader side won’t match the CRC attached to the packet on the writer side). If CRCs of both writer and reader match, the packet is considered free of errors. CRC processes do not exclude the existence of inconsistencies in the data itself—reason why other types of packet validation tests are computed—but it ensures a low likelihood of receiving unhealthy packets.

There are many different types of CRC processes one could opt for, some more adapted to a particular kind of application than others. The micro-controllers being used mount a **built-in hardware-based CRC calculator** (whose settings are not editable), which allows to perform CRC computation without the use of CPU resources and much faster. Otherwise, CRC computation uses a lot of resources, be it memory resources through the use of lookup tables, or computational resources through direct CRC computation. The particular settings of the built-in CRC generator are described on Table 7-5.

***Table 7-5****. Generation settings of the CRC used.*

|  |  |
| --- | --- |
| Parameter | Value |
| Name | CRC-32/MPEG-2 |
| Polynomial | 0x04C11DB7 |
| Init value | 0xFFFFFFFF |
| Reverse In | No |
| Reverse Out | No |
| Xor Out | 0x00000000 |

It is important to note that if the CRC was to be computed for a value non-multiple of 4 (bytes), this value should be right fill (i.e., Least Significant Byte (LSB) side) with as many 0x00 bytes as required to make the number of bytes input of the CRC multiple of 4. An example of the latter is provided below (values are given in hexadecimal and the sign describes an Xor binary operator).

Custom CRC computation algorithms can be tested on [www.crccalc.com](http://www.crccalc.com).

### Consistent Overhead Byte Stuffing (COBS).

So far both writer and reader are capable of sending packets, determining whether the packers have been corrupted during transmission, and understating what the transmitted information contained in healthy packets means. But a procedure is still required to determine when a new packet starts and when it ends while, at the same time, allowing the reader to resynchronize itself with the flow of upcoming information in the event that corrupted packets are detected. These issues can all be solved with a **stuffing procedure**.

Stuffing is a technique that consists in modifying what is to be sent in order to minimize the likelihood of appearance of the issues just mentioned. Stuffing can be done at the bit level to ease byte recognition (this is, for example, performed in the UART protocol, where the useful bits are simply enclosed between start and stop bits), or at the byte level to ease packet recognition. At this level, most techniques choose a start and stop byte and add these to the beginning and end of a packet. If the selected start/stop byte is already present in the packet, the affected bytes are modified by the stuffing algorithm. The goal of stuffing is to enclose the packet in between two bytes that are unique within the packet.

EX1 internal communication protocol makes use of a stuffing method called **Consistent Overhead Byte Stuffing (COBS)** created by Stuart Cheshire in 1997[[3]](#footnote-2). Its main advantages includes its simplicity, the fact that it only adds 2 (3 in our case[[4]](#footnote-3)) framing bytes to the packet (i.e., *overhead*) independently of the content of the message for packets shorter than 254 bytes, and it guarantees resynchronization if a corrupted packet is received.

The COBS process consists in using a 0x00 byte as the framing character and encoding each 0x00 byte not based on itself but based on all the following characters until the next 0x00 byte. How this works will become clear by looking at the example presented in Table 7-6.

***Table 7-6****. Example of COBS stuffing steps.*

|  |  |
| --- | --- |
| Data | Step |
| 07 09 00 01 00 00 02 03 04 05 06 00 18 22 | Initial data (hex.). |
| 07 09 00 01 00 00 02 03 04 05 06 00 18 22 **00** | Add a 0x00 byte at the end. |
| 07 09 00 01 00 00 02 03 04 05 06 00 18 22 00 | Delimit the blocks ending in 0x00. |
| **03** 07 09 **02** 01 **01** **06** 02 03 04 05 06 **03** 18 22 | Replace every block by a new block that starts with a byte describing its length and the remaining of the non-zero bytes. |
| **00** 03 07 09 02 01 01 06 02 03 04 05 06 03 18 22 **00** | Frame the packet with 0x00 bytes. |

An example of how the reverse process will work ( “unstuffing”) with the previously stuffed packet is shown in Table 7-7.

***Table 7-7****. Example of COBS unstuffing steps.*

|  |  |
| --- | --- |
| Data | Step |
| 00 03 07 09 02 01 01 06 02 03 04 05 06 03 18 22 00 | Initial data (hex.). |
| 03 07 09 02 01 01 06 02 03 04 05 06 03 18 22 | Remove framing bytes. |
| **03 07 09** 02 01 01 06 02 03 04 05 06 03 18 22 | Find the first block using its size as a ref. |
| 03 07 09 02 01 01 06 02 03 04 05 06 03 18 22 | Find the second and all subsequent blocks using the size bytes as ref. |
| 07 09 **00** 01 **00** **00** 02 03 04 05 06 **00** 18 22 **00** | Remove the first byte of each block and add a 0x00 byte at the end instead. |
| 07 09 00 01 00 00 02 03 04 05 06 00 18 22 | Remove the ending 0x00 byte. |

The stuffed size of each command and transmitted information packet is presented in Table 7-8.

***Table 7-8****. Stuffed size of each command and data transmitted.*

|  |  |  |  |
| --- | --- | --- | --- |
| Name | LEN | Packet size [bytes] | Stuffed packet size [bytes] |
| LIGHTS\_ON | 0 | 8 | 11 |
| LIGHTS\_OFF | 0 | 8 | 11 |
| SETSPEED\_FRONT | 4 | 12 | 15 |
| SETSPEED\_REAR | 4 | 12 | 15 |
| SETSTEER\_FRONT | 4 | 12 | 15 |
| SETSTEER\_REAR | 4 | 12 | 15 |
| VAL\_SPEED\_FRONT | 4 | 12 | 15 |
| VAL\_SPEED\_REAR | 4 | 12 | 15 |
| VAL\_ANG\_FRONT | 4 | 12 | 15 |
| VAL\_ANG\_REAR | 4 | 12 | 15 |
| VAL\_CURR\_FRONT | 2 | 10 | 13 |
| VAL\_CURR\_REAR | 2 | 10 | 13 |
| VAL\_ANG\_RCK | 4 | 12 | 15 |

### Automatically executed routines.

Now that all the fundamental blocks of code are ready, they need to be assembled into the main software. But we first have to consider one last thing: **prioritization of tasks**. Some tasks, such as control-related tasks, are much more important than others, such as communication-related tasks. In order to avoid control instabilities, a relatively high control frequency is needed. Whereas communicating commands from high level and executing them at low ground speed allows for some delay in the processing of the instruction.

In order to build some degree of prioritization into the system, extra timers were used, raising interrupters at a chosen frequency. Those interrupters then call the appropriate automatic control functions, even if the CPU is handling some other task. One has to be careful with this method, of course, as using an extremely high rate or an excessively long function can result in an overloaded CPU with no spare time to execute other tasks or even finish the primary tasks.

The tasks that are automatically executed are summed up in Table 7-9.

***Table 7-9****. Automatically executed tasks and their execution time.*

|  |  |
| --- | --- |
| Rate (Hz) | Task |
| 100 | Compute speed of the FDM. |
| 100 | Compute speed of the RDM. |
| 100 | Apply speed control command for the FDM. |
| 100 | Apply speed control command for the RDM. |
| 100 | Apply position control command for the FSM. |
| 100 | Apply position control command for the RSM. |
| 5 | Send current through each DM. |
| 5 | Send speed of each DM. |
| 5 | Send position of each SM. |
| 5 | Send rocker angle. |

As previously mentioned, the protocol has to ensure that a transmission rate of 92,160 bits/s is never exceeded in order not to overload the ring of the buffer. Based on Table 7-8, and assuming we are sending all the available information once every time, **the maximum sending rate at which all the information can be sent in series is approximately 912 Hz[[5]](#footnote-4)**.

By choosing 5 Hz for data transmission, we are sure that the transmission line will never be overloaded. If later on, 5 Hz is discovered to be insufficient to meet new data needs, the code should be simple enough to modify.

1. efficiency= No. of useful bits/Total bits = 8 data bits/(8 data bits + 1 stop bit + 1 start bit) [↑](#footnote-ref-0)
2. Max. baud rate = efficiency x baud rate [↑](#footnote-ref-1)
3. http://www.stuartcheshire.org/papers/COBSforToN.pdf [↑](#footnote-ref-2)
4. In the original COBS algorithm, only one single 0x00 bytes is added at the end. In our case we decided to also add one at the beginning to facilitate the detection of a starting transmission. [↑](#footnote-ref-3)
5. Max rate = max baud rate / SUM of stuffed size of packets = 92,160/101. [↑](#footnote-ref-4)