# Take control of your MINDSTORMS™ bricks

*by Oton Ribić*

Having assembled and sent our first messages to an EV3 brick last time, and making that motor finally turn, now it is about time to expand on this concept and establish a reliable communication sequence.

## *Replies and synchronization*

Upon each message successfully sent to the EV3 Smart Brick, it will always reply, even if the message's instructions cannot be obeyed or they contain errors. This reply gets sent only after the instruction to be done by the EV3 is fully completed. This is the key to establishing a solid synchronized communication, i.e. ensuring that one instruction gets completely done before another one begins, which is essential in practice.

Replies from EV3 are received through a dedicated serial port as well, the one designated for incoming traffic (check the first article in the series to remind yourself what this is about). What you have got to do through your code is send a message to the EV3 Smart Brick and then, depending on your implementation, either wait until the computer gets notified (called back) that some data awaits reception, or keep checking repeatedly if there is any data to be read. Then get the message proper from the serial port, and interpret its contents if some value has been asked for, e.g. in the case of reading sensor values. Afterwards, the entire process can be repeated.

These replies by the Smart Brick follow the same structural rules as the messages send to it — the first two bytes indicate its upcoming size, and the rest is the message payload. We won't go into analyzing all possible replies yet as that would

| BYTE NO. | 00 | 01 | 02 | 03 | 04 |
|---|---|---|---|---|---|
| **BYTE VALUES** | **03** | **00** | **00** | **00** | **02** |
| DESCRIPTION | Length | | Message | | OK |

expand this edition into an encyclopaedia, but let's look at how the standard "Done, everything OK" message from the EV3 looks like.

So, it has a total length of 5 bytes, among which the first two indicate the remaining size of 3 bytes (in reverse!). Then, the next two indicate the message ID it refers to having been completed, and finally the value 02 confirms everything has been done. Remember, if there are multiple messages sent asynchronously, then the Message ID it refers to is useful — but if we follow the strict synchronized principle, then it can and will always only refer to one message in its "inbox", whose ID we had previously set to zero. Or, in two-byte chunks: 00, 00.

To sum up in general: unless you're asking for the EV3 to provide some value back, which we will be covering later, it will reply with 3, 0, 0, 0, 2 as a confirmation that everything is OK, or something else if some error happened. If you want to learn more about these error messages and interpret them, check here: http://ev3.fantastic.computer/doxygen-all .

## *Encoding values*

Before we dive into more complex messaging next time, there is one necessary digression to make, regarding the system used for sending to and receiving numeric values from the EV3 Smart Brick. As long as we are dealing with small values, such as percentage of power to be used on a motor that needs to be rotated, it is simply directly encoded as a byte value. One of the examples was in the previous instance where we had converted the number 75 directly into its byte value, and sent it off packaged in the message.

The approach is different, however, when dealing with numbers that specify more complex parameters or larger values. The main example we will face is making the motor turn a specified number of degrees, i.e. a given angle. This value may be much too large to fit into one byte — even one full motor turn, of 360 degrees, would be too large to fit into one byte. Hence, EV3 uses C structures, specifically, 4-byte floating point C structures to represent numerical data.

Unless you're an experienced programmer, that probably doesn't mean a lot, but let's put it this way: there are various systems for encoding values into chunks of zeroes and ones, and decoding them back to "proper" numbers. LEGO® engineers chose that one among them, and unless you're working with C or some similar language where this is supported natively, it will be your code which needs to perform this encoding step. Fortunately, this is a fairly standard thing, and all popular languages have some way to elegantly perform it. E.g. if you are using Python, a struct library is readily available as a part of the standard package. (If you want to go into detail, you'll need struck.pack('f',value) function.)

In any case, a couple of queries on Google, with keywords for "C structs, number conversion" and your chosen language, should set you on the right track. Just make sure you're interpreting the number as a floating-point number, even if the value you are using does not need a decimal point. If you want to test whether your conversion works right, here are some examples; 360 should convert to 0, 0, 180, 67, and -360 to 0, 0, 180, 195. Zero converts to four zeroes, and 12.34 to 164, 112, 69, 65. This conversion will be essential for any further work, so make sure you've got it working well before proceeding.

And speaking of proceeding — we will use these very values in the next edition, where we will combine the knowledge from the previous articles and this one to fire off some more complex commands, such as controlled motor movements. Stay tuned!
#