Take control of your MINDSTORMS bricks (5)

by Oton Ribić

Finally, the combined knowledge we have collected throughout the previous four articles should now enable us to reach the ultimate goal we strove for: sending commands to our EV3 smart bricks via Bluetooth. Of course, the handiest and the most frequently used type among them is performing controlled motor movements, and that is exactly what we are going to do.

Assembling the message components

Without trying to explain again each of the command message components, let's just go ahead and work on an example. Let's suppose we want to turn the motor connected to the port no. 2 for one and a half turn (540°) in positive direction at three quarters of full speed, i.e. at the throttle of 75%. And build the message bit by bit.

1) Header. Firstly we have got to set up a header of the message, as discussed in the previous articles. Since we are not going to use it for anything "fancy" this time, we will just begin with five zeroes, i.e. five bytes with values of zero.

2) Direction. Now we will want to set the motor direction for this command. This is done by adding bytes 167, 0, then the number of the motor which is 2 in our case, and finally 1 for forward, or 63 for reverse direction. So we have got 167, 0, 2, 1 here.

3) Movement values. Then comes the main part, the movement instruction itself. It begins with 174, 0, then continues with the number of the motor, again 2. Next is the speed: it is the value 129 followed by the desired speed which yields 129, 75. Next is a rampup value for which we can use zero encoded to five bytes, which is 131, 0, 0, 0, 0. Then finally the angle we intend to turn, 540 encoded in five-byte structure, which is 131, 28, 2, 0, 0. Then the rampdown value which is again zero, turning out to 131, 0, 0, 0, 0. Finally, the parameter that says to brake once completed, which is a simple final byte 1 for this section.

4) Instruction to start. Having set all the parameters, we will now add the instruction for the motor to actually start this meticulously prepared work. It is rather simpler: 166, 0, followed by the number of the motor, which is 2.

5) Wait for completion. If we want the EV3 to perform the movement fully before going onward to the next one, we will now add 170, 0, 15, which is essentially "wait for completion". Without it, the next instruction will begin while the motor turns, which you may want, or may avoid.

6) Length. Let us finally count the length of the message we have assembled: it contains 36 bytes. So we put 36,0 in front of it.

If everything went well, we got the following message and structures.

36, 0, 0, 0, 0, 0, 0, 167, 0, 2, 1, 174, 0, 2, 129, 75, 131, 0, 0, 0, 0, 131, 28, 2, 0, 0, 131, 0, 0, 0, 0, 1, 166, 0, 2, 170, 0, 15

And each part corresponds to its own purpose in matching colors: Length, Header, Direction, Specifying motor, Speed, Rampup, Amount to turn, Rampdown, Brake when completed, Begin rotating, Wait for completion.

Sending the message

At this point we are ready to "fire" these final 38 bytes to the EV3 brick via the virtual serial port encapsulated within the Bluetooth protocol. As explained in the previous articles, the EV3 brick will acknowledge receipt of this message 0 when finished. We can then send further messages if we desire so.

If you want to go to the full lengths and implement the encoding of the values into five-byte structures, here are the values for each byte. This assumes that the ampersand (&) is used as a binary AND operator, and >> a binary shift to the right by a given number of places. (This works "as is" in Python.)

```
byte1 = 131
byte2 = value & 255
byte3 = (value >> 8) & 255
byte4 = (value >> 16) & 255
byte5 = (value >> 24) & 255
```

Of course, this works only with integers, but that is anyway the underlying assumption for this entire tutorial. If you will be working with divisions of numbers, it is always a wise precaution to round any numbers entering this calculation down into integers.

Rotating multiple motors at once is done by simply constructing and firing away independent messages, one for each motor, and not enabling the "wait for completion" parameter in them. That way all the rotations will start and proceed simultaneously.

However, at this level there is no simple and foolproof way to control the position of each motor at every particular moment. I.e. if we start one motor at full speed and the other one at half speed, it is only an optimistic assumption that, at any given moment afterward, the latter will have done exactly half the movement of the former.

If you're after very accurate simultaneous movements, e.g. for drawing a diagonal line with an X-Y plotter, consider splitting the movements into smaller segments, and using lots of reduction to further iron out any differences between the motors. Of course, the price to pay in this case is slower execution, so you will have to find the formula that works the best for you.
#