Take control of your MINDSTORMS bricks (3)

Text and images by Oton Ribić

If you have been with us for the two previous articles, this time is where the fun part begins: assembling our first 'real' messages to be sent to the pBrick to order the performance of an action. For now, we will stay focused on the EV3 system, whereas the NXT (which is a tad more complex) will be covered later.

As explained previously, there is a standard 'header' for each message, i.e. a section that announces how long the message will be, along with some other parameters. This time, let us go further and construct the body of our message for an EV3.

Initial parts

Let us begin by trying to rotate a motor, which is probably the most common operation anyway. There are various ways to control a motor, including specific steps, how much to turn, at which speed and power, ramp-up, coast, etc. — but there is also a much simpler operation, involving just making a motor turn indefinitely. That will be our starting point.

All the bodies of such messages are constructed by specifying a set of variables, i.e. parameters required for the operation — such as speed and motor in this case, and other data for other types of jobs — and then sending the instruction to make it act on the received data. In our first case, we need to specify three parameters: which motor we intend to turn, at what power or speed, and in which direction. Let us try with the first motor, at 75% power, in a positive direction.

First we need to calculate the code of our motor, as it keeps being used throughout the message. It is a rather straightforward job of exponentiating 2 to the power of the motor number, with the first one being zero. So, the first motor's code is 1, the second's 2, then 4 and finally 8 for the fourth.

This can be used to assemble the first part of our message, featuring polarity. If we want to turn the first motor in the positive direction, our message will specify this with bytes: 167, 0, 1, 1. The 167, 0 is fixed; the next number is a code for the motor, and the final 1 specifies positive direction. Otherwise, the final byte would have been 63.

EV3 positive spin direction

You may be asking yourself — why these codes exactly? Why 167 of all numbers? Well, the simplest and the most straightforward answer is that this is the 'language' EV3 uses to communicate. Though it may seem arbitrary, there is an underlying logic which supports it, though getting deeper into it would expand this article beyond tolerability, so let's for now just stick with the numbers we have.

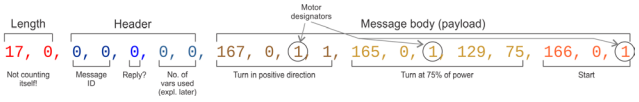
So, our message starts with 167, 0, 1, 1. Let's continue by specifying the speed we want it to turn at. The bytes we need are 165, 0, 1 (again the motor), 129, and finally, the power percentage as a number — in our case, 75. And then, on we go with the final instruction to 'make it happen': 166, 0, 1.

For the assembly!

All right, we've collected all we need and we can start building our message. Let's begin by concatenating these fragments in the aforementioned order, to get 167, 0, 1, 1, 165, 0, 1, 129, 75, 166, 0, 1. This is the body of our message containing three segments, specifying the direction of rotation, its desired speed, and the instruction to start.

But, remember from the last time: we have to add a header to this body to construct a final, proper message which the EV3 can receive and interpret. There are two bytes which specify the number of this particular message, so that the EV3 can refer to it when replying. We can freely keep them at 0, 0 for this exercise. And the number of globally used variables is also 0. So let's add three more zeros to the front of our message body.

And then there is the final step: in order to be sure the message has been received properly, the EV3 pBrick needs to know how long it is. So let us count the number of bytes the final message contains, to come up with the figure 17. We have to represent this as a two-byte number in front of the entire message, with the smaller byte (properly said in engineering: least significant) being first. So the length designator (which doesn't count itself!) will turn out to be 17, 0. This makes our final message look like this: 17, 0, 0, 0, 0, 0, 0, 167, 0, 1, 1, 165, 0, 1, 129, 75, 166, 0, 1.



Just go ahead and send these bytes to the serial port you have created for the purpose and connected the pBrick to, according to the data from our previous articles, and the first EV3 motor should start spinning! Of course, the question of how you prefer to send these bytes over to the pBrick is entirely down to you, i.e. down to your choice of the language or system you prefer using. C and its derivatives mostly have native support for writing directly to ports; Python has an excellent PySerial library; most at least half-decent languages have at least one way to do so. Don't worry, serial ports are such a stable and mature technology, that there is no question of your system not supporting it.

Keep in mind that this instruction just says 'start spinning' without any time or turn limit. So it will indeed keep turning until told otherwise or (obviously) turned off. In the future we will dig a bit deeper into a more complex form of the turning instruction, which lets the user control exact movements and parameters.

But before that, we will have to examine how the pBrick has actually replied while obeying the above messages. Although it is not always mandatory, any kind of controlled movement is unthinkable without monitoring the pBrick's replies and acting upon them. So stay tuned!
#