

Controlling Robots Built with the LEGO[®] MINDSTORMS[®] NXT Brick

Denis Cousineau

The NXT is a general-purpose processor that can be used to control motors and sensors; it is ideal for building autonomous robots. It can also communicate with more elaborate software located on a computer, using a Bluetooth communication port. In this article, we show how to communicate with the NXT by sending the correct bytes. We also introduce a package that manages all the exchanges through functions. These functions can be used in conjunction with dynamic cells to display the robot's status and control the robot's motor.

■ Introduction

Robots are ideal vehicles for testing cognitive theories regarding learning, adaptations, and classification of environmental stimuli. Most robots are built around a central processor in charge of dealing with the motor output and the sensory input. To that end, in 2006 LEGO released a new programmable brick, the LEGO MINDSTORMS NXT. This brick, equipped with four sensor inputs and three motor outputs, has 256 kilobytes of flash memory in which files can be stored. It can run programs compiled for the brick (files ending with the .rx extension and compiled for this system using LEGO LabVIEW or third-party compilers such as NBC or NXC). The versatility and low cost of this brick has made it an ideal vehicle for developing robotic projects.

The NXT is well-suited for developing autonomous robots such as scouting robots. Learning algorithms can also be tested; for example, to find the exit of a maze more aptly than by trial and error. Finally, the NXT can also be used to study social cognition models, in which multiple robots must interact to enhance the group survival rate.

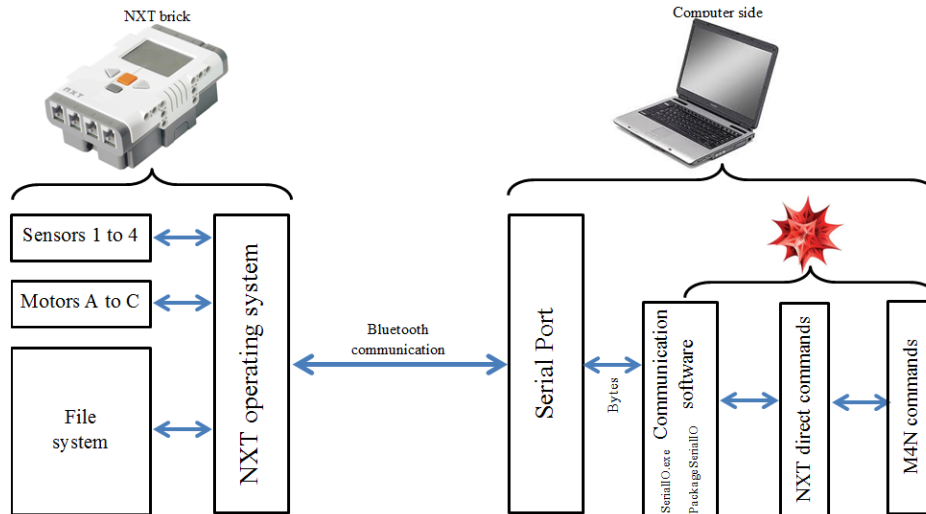
This article shows how to control the brick over a Bluetooth communication link. We also introduce the *Math4NXT* package, which makes the task easier. Used in conjunction with *Mathematica's* built-in `Dynamic` command, controlling a LEGO robot becomes very simple.

■ Establishing a Communication Link

The first time you start the NXT brick, use your computer's Bluetooth tools to look for the brick and establish a communication link. On that very first use, you have to provide a passkey on the brick (default is "1234") and on the computer. Then you need to choose the "Dev" service available on the brick. At that point, a virtual serial communication port (a COM port) is allocated to serve as a communication link between the computer and the brick. The exact COM port given is variable (e.g. COM11) but remains constant for this NXT-computer pair (detailed instructions to achieve the pairing can be found in [1]). Once you know which COM port to use, you do not need to repeat the process.

The NXT brick has an operating system (firmware) that immediately handles requests received from the COM port. It does so even if the brick is currently running a program. These commands are called *direct commands* in the LEGO documentation [2]. Direct commands include file manipulation, motor control, sensor configuration and reading, and communication between bricks. Commands are sent over the COM port as *telegrams* composed of a succession of bytes (numbers between 0 and 255). For some commands, a reply can be returned by the brick, also in the form of a telegram. Each telegram is preceded by the length of the telegram, given as a WORD number, that is, using two bytes. Hence, controlling the NXT brick is simply a matter of sending the correct bytes in the correct order over a serial port.

In order to use a serial COM port, use the *Mathematica* package called SerialIO [3]. This package has two parts. The first is an executable program, suited for your computer's processor and operating system. SerialIO provides such programs for Linux and Windows OS (both 32 and 64 bit) as well as for Mac OS X. The second part has the *Mathematica* functions that can be used in your project. It includes the commands `SerialOpen` and `SerialClose` to open and close a communication link on a given COM port, and `SerialWrite` and `SerialRead` to send or receive bytes from the COM port. The following figure illustrates the different layers through which information goes during a communication between an NXT brick and a computer.



▲ **Figure 1.** The software architecture by which an NXT brick and a computer can communicate. The last three boxes on the right represent *Mathematica* programs, and the last two are given by the package *Math4NXT* described next.

Get the package `SerialIO` from Wolfram Library Archive and install it in a folder of your choice (a good place is in the *Mathematica* AddOns folder, under the `ExtraPackages` subfolder). For convenience, you can set your working directory to this path so that the *Math-Link*-compatible program is located automatically using the following instruction.

```
SetDirectory[
  FileNameJoin[{$InstallationDirectory, "AddOns",
    "ExtraPackages", "SerialIO", $SystemID}]]];
```



Then load the package with `Needs`.

```
Needs["SerialIO`"]
```

The command is successful if you can see a process named `SerialIO` running in the background of your computer. The following command opens a COM port and therefore establishes a communication link with the brick (it must be turned on).

```
mybrick = SerialOpen["COM11"];
```

If you have an NXT brick and wish to evaluate the input cells in this article, select any input cell while holding the `Alt` key; that selects all input cells. Then from the *Mathematica* menu, choose `Cell ▶ Cell Properties ▶ Evaluatable`. Without an NXT brick, running these commands returns error messages or unevaluated input.

To see if the connection is open, look at the brick LCD display in the upper-left corner. If you see , it means the connection is not established. However,  means that the connection is established. The variable `mybrick` contains the stream descriptor of the COM port and is used in all subsequent communications. Closing the COM port is done with the following instruction.

```
SerialClose[mybrick]
```

Exiting *Mathematica* (or ending the kernel) also closes the COM port and purges the process `SerialIO` running in the background.

■ Sending a Telegram and Getting the Reply

The `SerialIO` package was made to send text over the COM port (either individual characters or strings of text). However, for controlling the NXT, it makes more sense to send numbers directly. Hence, we need to convert numbers to the corresponding letters in the ASCII code before sending them. This is a bit awkward and we propose a neater solution later.

A telegram always begins with one byte indicating whether the NXT should return a reply telegram or not. This is your choice, but many commands sent to the brick only return a status byte (0 for success, or an error message). In the case of a status byte, the reply telegram is not useful. To ask for a reply, the first byte must be 0; 128 means that the NXT must not send a reply.

The second byte of a telegram is always the command number. These numbers are given in the LEGO Group documentation [2]. The subsequent bytes depend on the command sent.

For example, asking the NXT to play a sound is achieved by the direct command `PlayTone`. Its command number is 3. Then two parameters must be provided: the tone frequency (a number between 200 and 14000) and the tone duration (in milliseconds, a number between 0 and 65535). These two parameters are coded as `UWORD` (unsigned `WORD`), that is, over two bytes, the least significant byte first. For convenience, we write a conversion function `toUWORD`.

```
toUWORD[value_] := {Mod[value, 256], Quotient[value, 256]}
```

Hence, a tone frequency of 480 Hz for a duration of two seconds corresponds to the following bytes.

```
toUWORD[480]
toUWORD[2000]

{224, 1}

{208, 7}
```

The whole telegram would then correspond to the following.

```
telegram = {0, 3, 224, 1, 208, 7};
```

It must be preceded by the telegram length, obtained with the following instruction.

```
telegramlength = toUWORD[Length[telegram]]

{6, 0}
```

We concatenate the two parts to get a complete message, and get a list of the characters corresponding to the numbers given (those characters may not be viewable on your computer system) with this line of code.

```
message = Map[FromCharacterCode,
  Flatten[{telegramlength, telegram}]]

{!, , , !, à, , Ð, !}
```

Let us send those characters to the NXT brick, one by one.

```
Map[SerialWrite[mybrick, #] &, message]

{!, , , !, à, , Ð, !}
```

You should hear a tone now. Because we asked for a reply (the first byte was zero), we need to read it.

```
reply = SerialRead[mybrick]

!!!
```

Again, the reply is composed of text (the characters of which may not be easily viewable). Let us convert them to a list of numbers.

```
ToCharacterCode[reply]
```

```
{3, 0, 2, 3, 0}
```

The bytes returned are, in order: the length of the returned telegram, coded over two bytes in UWORD (here 3, 0 means a telegram of length 3); the number 2 indicates that this is a reply telegram; the number 3 indicates the command number that issued a reply (here, 3 is the number for the command PlayTone); and finally, the status of the command (where 0 indicates a success). The command PlayTone does not return other information; some direct commands may return more complex telegrams.

As an example, we send the command GetBatteryLevel. This command has number 11 and requires no parameters (no extra information). If a reply is asked for (but it is pointless to send this command and not ask for a reply), it returns the status of the command (0 for success) followed by two bytes indicating the voltage of the batteries on the NXT in millivolts. To decipher the voltage, let us create another conversion function, fromUWORD.

```
fromUWORD[bytelo_, bytehi_] := bytehi * 2561 + bytelo * 2560
```

Let us assemble the telegram, concatenate the telegram length, convert this to a list of characters, and send them.

```
telegram = {0, 11};  
telegramlength = toUWORD[Length[telegram]];  
message = Map[FromCharacterCode,  
  Flatten[{telegramlength, telegram}]];  
Map[SerialWrite[mybrick, #] &, message]
```

```
{!, , , !}
```

Now, let us read the reply and convert it to numbers.

```
reply = SerialRead[mybrick] // ToCharacterCode
```

```
{5, 0, 2, 11, 0, 247, 27}
```

The first two bytes are the reply length (5 bytes); 2 and 11 indicate that this is a reply for a `GetBatteryLevel` command; 0 indicates that the command was successful; and finally, 247 and 27 indicate the battery voltage, which we convert to a number with `fromUWORD`, defined above.

```
fromUWORD[247, 27]
```

```
7159
```

Hence, the batteries convey 7.16 volts, much below the expected 9 volts for fresh batteries.

■ Override `SerialWrite` to Send an Integer or a List of Integers

Because it is more convenient to think in terms of numbers sent to the brick rather than characters, we expand the command `SerialWrite` contained in the package `SerialIO` to include the possibility of sending individual integers between 0 and 255 (bytes), and also to send a list of bytes.

```
ByteQ[x_] := 0 ≤ x ≤ 255
```

```
SerialWrite[port : SerialPort[_String, _Integer],  
  datum_Integer] :=  
  SerialIO`SerialWrite[port, FromCharCode[datum]  
] /; ByteQ[datum]
```

```
SerialWrite[port : SerialPort[_String, _Integer],  
  data_List] :=  
  Map[SerialIO`SerialWrite[port, #] &, data  
] /; And@@Map[ByteQ, data]
```

With this expansion, a whole message (composed of bytes) can be sent in one call to `SerialWrite`. For example, the following instructions assemble again the `PlayTone` command (this time without a reply).

```
tone = 480;  
duration = 2000;  
telegram = {128, 3, toUWORD[tone], toUWORD[duration]} //  
  Flatten;  
telegramlength = toUWORD[Length[telegram]];  
message = Flatten[{telegramlength, telegram}]  
  
{6, 0, 128, 3, 224, 1, 208, 7}
```

The whole message is sent with one command call.

```
SerialWrite[mybrick, message];
```

■ The Direct Commands Provided in *Math4NXT*

The above shows how any command can be sent to the NXT brick and its reply read. Afterward, it is simply a question of assembling a telegram in the right order with the right information, and—if asked for—reading the reply and segmenting it into the correct information. To that end, the documentation provided by the LEGO Group ([2], particularly the appendix 2) is quite complete.

To make things simpler, we coded all the direct commands in a package called *Math4NXT*. First get the package *Math4NXT*. Since it uses the package `SerialIO` in the background, also install this package in a path that is searched by *Mathematica* (e.g. `FileNameJoin[{$InstallationDirectory, "AddOns", "ExtraPackages", "SerialIO"}]`). Then load *Math4NXT* using the following instruction.

```
Needs["Math4NXT`",  
  FileNameJoin[$HomeDirectory, "Documents", "Binaries",  
    "Math4NXT.m"]]
```

```
Math4NXT loaded.
```

```
Do ? Math4NXT`* to see all the functions available  
or ? NXT* for the direct commands  
and ? M4N* for the higher-level commands.
```

Since we did not place the package *Math4NXT* in a folder searched by *Mathematica*, we provided its path in the `Needs` instruction. Then open the serial port (if this was not already done) as before.

```
mybrick = SerialOpen["COM11"];
```

All the direct commands begin with the letters `NXT`. Hence, to read the battery level, the command in the package is called `NXTGetBatteryLevel`.

```
? NXTGetBatteryLevel
```

`NXTGetBatteryLevel[port]` returns the battery level (mV). Default option is `Reply→True`.

```
res = NXTGetBatteryLevel[mybrick]
```

```
{Status → 0, VoltageMV → 7062, Percent → 78.4667%}
```


By default, the command returns the answer using named strings (e.g. "Status"). You can extract one piece of information as usual.

```
"VoltageMV" /. res
```

```
7062
```

This format can be changed, for example to raw (i.e. bytes as in the previous two sections), using the option `ResultFormat`.

```
NXTGetBatteryLevel[mybrick, ResultFormat → Raw]
```

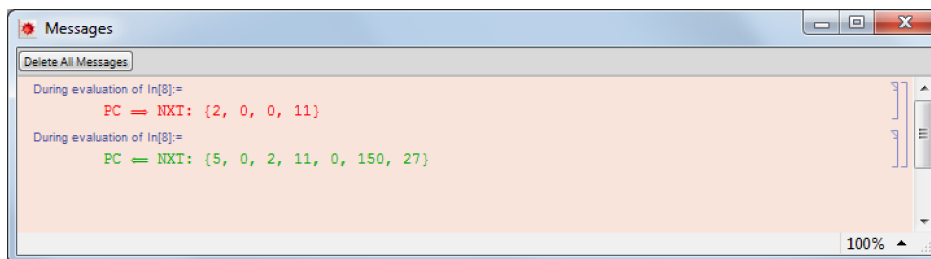
```
{5, 0, 2, 11, 0, 178, 27}
```

Alternatively, the bytes sent and received can be displayed in a separate window using the option `Echo` → `True`.

```
NXTGetBatteryLevel[mybrick, Echo → True]
```

```
{Status → 0, VoltageMV → 7062, Percent → 78.4667%}
```

This option opens the Messages window.



▲ **Figure 2.** The Messages window opens automatically when the option `Echo` → `True` is used.

All the direct commands are accessible.

? **NXT***

▼ Math4NXT™		
NXT	NXTGetVolume	NXTRequestFirstModule
NXTClose	NXTKeepAwake	NXTRequestNextModule
NXTCloseModule	NXTLSGetStatus	NXTResetInputScaledValue
NXTDelete	NXTLSRead	NXTResetMotorPosition
NXTFindFirst	NXTLSWrite	NXTSetBrickName
NXTFindNext	NXTMessageRead	NXTSetInputMode
NXTGetBatteryLevel	NXTMessageWrite	NXTSetOutputState
NXTGetCurrentProgramName	NXTOpenRead	NXTSetVolume
NXTGetDeviceInfo	NXTOpenWrite	NXTStartProgram
NXTGetFirmwareVersion	NXTPlaySoundFile	NXTStopProgram
NXTGetInputValues	NXTPlayTone	NXTStopSound
NXTGetOutputState	NXTRead	NXTWrite
NXTGetSleepTime	NXTReadIOMap	NXTWriteIOMap

■ Extending the Direct Commands

The direct commands have many limitations. For one, due to the nature of the COM port, messages cannot be longer than 253 bytes. This is a big limitation when dealing with files that can be much longer. Also, the LEGO sensors all work in very different ways; some sensors are passive, not requiring power, while others are active, and still others are programmable, like the I2C sensors. To make use of the commands more uniformly and to avoid the length limitation, we programmed higher-level commands that deal internally with these difficulties.

The names of all of these commands begin with M4N.

? **M4N***

▼ Math4NXT			
M4NBrakeMotor	M4NInitialize	M4NRunMotorFor	M4NStopMotor
M4NDownloadToNXTFile	M4NMotorFreeQ	M4NSearchModules	M4NUploadFromNXTFile
M4NFileExistsQ	M4NReadSensor	M4NSetMotor	
M4NFileNames	M4NRunMotor	M4NSetSensor	

One example is the command `M4NFileNames` that lists all the files present on the brick (optionally with the file size).

? **M4NFileNames**

`M4NFileNames[port,filestem]` lists all the files on the NXT with a filename corresponding to `filestem`. Wildcards are allowed. Default option is `FileDetails→None`.

```
M4NFileNames[mybrick, "*.*", FileDetails → All] //  
TableForm
```

```
MotorControl22.rxe    37 530  
! Startup.rso        4084  
! Attention.rso      881  
! Click.rso          229  
Try-Color.rtm        4346  
Try-Touch.rtm        1238  
Try-Light.rtm        684  
Try-Ultrasonic.rtm  1208  
Try-Motor.rtm        676  
Try-Sound.rtm        638
```

This command works by calling the direct commands `NXTFindFirst` (command number 134), `NXTFindNext` (135), and `NXTClose` (132), using the algorithm given in Program 1. To see the multiple calls to NXT commands, use the option `Echo → True` in `M4NFileNames`.

```
res=NXTFindFirst[mybrick,"*.*"];
While [{"Status"/.res]==0,
  res=NXTFindNext[mybrick,"Handle"/.res]
]
NXTClose[mybrick];
```

- ▲ **Program 1.** General algorithm used by `M4NFileNames` to retrieve all the file names present on the NXT brick. This algorithm does not show how to gather the file names (contained as bytes in `res`) into a list.

Likewise, to facilitate the use of sensors, a command is provided that informs *Mathematica* of the sensor types connected. Afterward, reading the sensor is done according to the type of sensor connected. To set the sensor type, use `M4NSetSensor`.

```
M4NSetSensor[mybrick, Sensor1 → TouchSensor]
```

After this, reading the sensor is done with a universal command, `M4NReadSensor`. To keep in line with the convention of direct commands, the first sensor is on the NXT input port 0.

```
M4NReadSensor[mybrick, 0]
```

```
0
```

The result is either 1 for “pressed” or 0 for “unpressed.” This command works the same way whatever type of sensor is connected. Hence, if you connect the ultrasound sensor on the third sensor input, you can issue the following two commands.

```
M4NSetSensor[mybrick, Sensor3 → UltrasoundSensor,
Echo → True]
```

```
M4NReadSensor[mybrick, 2, Echo → True]
```

```
6
```

The ultrasound sensor is a complex sensor aimed at detecting the distance of obstacles in front of it (in centimeters). It is equipped with a microprocessor that follows the I2C protocol. It must first be powered up and time given to it to boot up before a reading can be made. Using the option `Echo → True`, you see that much more exchange of information is needed to initialize an ultrasound sensor than to initialize a touch sensor. Yet, all these transactions are completely invisible with the `M4N` commands.

■ Working with Motors

Another limitation of the direct commands provided by the NXT's firmware concerns the motor control. The new LEGO motors contain tacho counters that keep track of the number of degrees of rotation executed. However, the direct command can only stop a motor once a target number of degrees is reached, so that in effect, the motor overturns.

PID control of a motor is a more effective way of controlling rotation. It sets a target number of degrees and adjusts the power of the motor so that it slows down when the target is almost reached (using integral and derivative; see [4, 5]).

The authors of [5] developed a PID controller for the NXT. This controller is located on the NXT brick; the computer sends a target movement, and the controller takes control of the motor, adjusting the power until the target number of degrees is reached. The controller program is called `MotorControl22.rxe`, for Version 2.2.

Get the program `MotorControl22.rxe` and transfer it to the brick. To do so, import the contents of the file into *Mathematica* and download this to the NXT. Once set (this step may take a few minutes over Bluetooth as the controller is 37 kilobytes long), start the controller. Here are the instructions to do so; the first part checks that the file is not already on the brick.

```
If [ ~ M4NFileExistsQ[mybrick, "MotorControl22.rxe"],
  content = Import [$HomeDirectory <> "MotorControl22.rxe",
    "Byte"];
  M4NDownloadToNXTFile [mybrick, "MotorControl22.rxe",
    content];
]
NXTStartProgram [mybrick, "MotorControl22.rxe"]
```

Finally, inform your program of what type of motors you have and to which port they are connected using `M4NSetMotor`.

```
M4NSetMotor [mybrick, MotorA → TachoMotor,
  MotorC → RegularMotor]
```

Incidentally, `M4NSetMotor` uploads `MotorControl22.rxe` if it is not present on the brick, and starts this program if it is not currently running. Therefore, you do not have to worry about these technicalities and can rely on `M4NSetMotor` alone.

Now that the controller is up and running, you can send orders to it using commands related to motors.

? M4N*Motor*

▼ Math4NXT		
M4NBrakeMotor	M4NRunMotor	M4NSetMotor
M4NMotorFreeQ	M4NRunMotorFor	M4NStopMotor

M4NRunMotor, M4NStopMotor, and M4NBrakeMotor can be used on regular motors as well as on tacho motors. However, M4NRunMotorFor can only be used with tacho motors and involves the PID controller. Once a motor is controlled by the PID controller, you should not send other instructions to it. Therefore, the predicate M4NRunMotorFreeQ can be used to check if the controller is done.

The following instruction starts the two motors.

```
M4NRunMotor[mybrick, {0, 2}]
```

Stop the motors using two different modes (M4NBrakeMotor locks the motor, whereas M4NStopMotor ceases to power the motor).

```
M4NBrakeMotor[mybrick, MOTORA]
M4NStopMotor[mybrick, MOTORC]
```

The following instructions check that motor MOTORA is free before a movement, during the movement (10 complete turns), and six seconds after the movement began.

```
M4NMotorFreeQ[mybrick, MOTORA]
M4NRunMotorFor[mybrick, MOTORA, MotorPower → 100,
  TachoLimit → 3600]
M4NMotorFreeQ[mybrick, MOTORA]
Pause[6]
M4NMotorFreeQ[mybrick, MOTORA]
```

True

False

True

■ Configuring a Robot

At the level of the program, a robot has a certain configuration, defined by what sensors are connected to the input and what motors are connected to the output. We showed two commands earlier that can be used to define the sensors and the motors.

```
M4NSetSensor[mybrick, Sensor1 → TouchSensor]
M4NSetMotor[mybrick, MotorA → TachoMotor,
  MotorC → RegularMotor]
```

We created a general command, `M4NInitialize`, whose purpose is to set all these items in one command. In addition, it can also be used to set the volume level and give a name to the brick (if you have many bricks, a good habit is to rename them by the COM port to which they are connected).

```
M4NInitialize[mybrick, Sensor1 → TouchSensor,
  MotorA → TachoMotor, MotorC → RegularMotor, Level → 1,
  SetName → "Com11"]
```

All systems ok!

In addition, `M4NInitialize` checks that the firmware version on the NXT is compatible with this package and checks that the battery has enough charge (it returns a warning if not). It is also possible to give the path of the folder containing the controller `MotorControl22.rxe` with the option `MotorControlPath` in case it is not already on the NXT.

■ Dynamic Control of the Robot

The M4N and NXT commands can be used dynamically. For example, to get the current state of the touch sensors, you can follow these steps.

First, initialize the sensors using `M4NInitialize` or `M4NSetSensor`.

```
M4NSetSensor[mybrick, Sensor1 → TouchSensor,
  Sensor2 → TouchSensor, Sensor3 → TouchSensor,
  Sensor4 → TouchSensor]
```

The `ShowSensor` function creates a panel with a title containing On or Off, based on the sensor value. This function uses some options for aesthetic purposes only.

```
ShowSensor[n_] := Panel [
  If[M4NReadSensor[mybrick, n - 1] == 1, "On", "Off"],
  "Sensor " <> ToString[n],
  BaseStyle → {FontColor → Red, FontSize → 36},
  ImageSize → 1. × 72, Alignment → Center]
```

Then, we can use `ShowSensor` four times in a row to get a row of indicators.

```
Dynamic[
  {ShowSensor[1], ShowSensor[2], ShowSensor[3],
   ShowSensor[4]}, UpdateInterval -> 0.1
]
```

The option `UpdateInterval` determines at what interval the sensors are consulted.

As a last example, we create a joystick that controls two motors. The speed of rotation of the motor (adjusted by the `MotorPower` option) depends on the up/down position of the joystick. In addition, the motors move synchronously if the left/right position of the joystick is centered.

We first define an extended `ClickPane` that responds to more events than the regular `ClickPane` function.

```
myClickPane[object_, opts___] := DynamicModule[{k1, k2, k3},
  Dynamic[
    EventHandler[object,
      "MouseClicked" -> (
        If[{k1 = CurrentValue["AltKey"]}, AltClick /. {opts}];
        If[{k2 = CurrentValue["ControlKey"]},
          CtrlClick /. {opts}];
        If[{k3 = CurrentValue["ShiftKey"]},
          ShiftClick /. {opts}];
        If[Not[k1] && Not[k2] && Not[k3], Click /. {opts}]
      ),
      "MouseDown" -> (
        If[{k1 = CurrentValue["AltKey"]},
          AltMovingMouse /. {opts}];
        If[{k2 = CurrentValue["ControlKey"]},
          CtrlMovingMouse /. {opts}];
        If[{k3 = CurrentValue["ShiftKey"]},
          ShiftMovingMouse /. {opts}];
        If[Not[k1] && Not[k2] && Not[k3], MovingMouse /. {opts}]
      ),
      "MouseUp" -> (
        If[{k1 = CurrentValue["AltKey"]},
          AltReleaseMouse /. {opts}];
        If[{k2 = CurrentValue["ControlKey"]},
          CtrlReleaseMouse /. {opts}];
        If[{k3 = CurrentValue["ShiftKey"]},
          ShiftReleaseMouse /. {opts}];
        If[Not[k1] && Not[k2] && Not[k3], ReleaseMouse /. {opts}]
      )
    ]
  ]
]
```


We must not forget to initialize the motors (so that the PID controller is turned on).

```
M4NInitialize[mybrick, MotorA → TachoMotor,  
MotorB → TachoMotor];
```

That is it. The following shows a panel with a red dot (the top of the joystick). It constantly sets the motor power to be proportional to the position of the joystick. In addition, if the joystick is released, it returns to the center with an exponential function implemented by the variable `mult`. Finally, if you click the pane anywhere, the joystick instantly returns to the center with a beep.

```
DynamicModule[{pt = {0, 0}, mult = {1, 1}},  
myClickPane[  
  Dynamic[  
    pt = pt × mult;  
    M4NRunMotor[mybrick, MOTORA,  
      MotorPower → Round[100 If[pt[[1]] < 0, 1 + pt[[1]], 1] pt[[2]]];  
    M4NRunMotor[mybrick, MOTORB,  
      MotorPower → Round[100 If[pt[[1]] > 0, 1 - pt[[1]], 1] pt[[2]]];  
  
    Graphics[{  
      Red, Disk[pt, 0.05],  
      Transparent, Rectangle[{-1.2, -1.2}, {1.2, 1.2}]  
    },  
    {ImageSize → {4 × 72, 4 × 72}, Frame → True}  
  ]],  
  MovingMouse :=> (mult = {1, 1}; pt = MousePosition["Graphics"];  
    pt = {Max[-1, Min[1, pt[[1]]], Max[-1, Min[1, pt[[2]]]}),  
  ReleaseMouse :=> (mult = {1 / 3, 1}),  
  Click :=> (pt = {0, 0}; Beep[])  
]
```

■ A Line-Follower Project

To illustrate a simple example of an autonomous robot, we program the line-follower robot. This robot's purpose is to follow the edge of a line. The line should be dark on a light background (or vice versa; the contrast along the edge is the important factor). To achieve this, you need a robot equipped in the front with a light detector (see, for example, the assembly instructions given on p. 33 of the MINDSTORMS education booklet [2]).

For this project, the configuration requires a light sensor and two motors, which we initialize.

```
M4NInitialize[mybrick, MotorA → TachoMotor,
  MotorC → TachoMotor, Sensor4 → LightActiveSensor]
```

All systems ok!

In a first phase, we need to calibrate the sensor so that the readings on light and dark surfaces are known. Use the following to average 100 readings while you move the robot over a white surface.

```
white = Table[M4NReadSensor[mybrick, 3], {100}] // N // Mean
```

68.01

Then use this while moving the robot over a dark surface.

```
black = Table[M4NReadSensor[mybrick, 3], {100}] // N // Mean
```

47.

The mean of the above two values is the critical value. For a robot that follows the left edge of a dark line, for readings lighter than the mean, we want the robot to steer to the right (and for readings darker than the mean, we want the robot to steer to the left).

```
mean = (white + black) / 2
```

57.505

The following short program moves the robot at a moderate speed (the variable `basespeed` is set to 20) and adjusts the base speed according to the difference between the current reading and the mean.

```
basespeed = 20; mult = 0.75;
M4NRunMotor[mybrick, {0, 2}, MotorPower → basespeed,
  SpeedRegulation → False]
Do[
  color = M4NReadSensor[mybrick, 3];
  M4NRunMotor[mybrick, 0,
    MotorPower → Floor[basespeed - mult (color - mean)]];
  M4NRunMotor[mybrick, 2,
    MotorPower → Floor[basespeed + mult (color - mean)]],
  {100}
]
```

```
M4NRunMotor[mybrick, {0, 2}, MotorPower → 0]
```

The constant `mult` is used to control the magnitude of the steering. Larger values result in a robot's making abrupt changes in direction. This quantity should be small for smooth movement. Yet, if there are abrupt curves in the trajectory, a small multiplier gives a robot that cannot turn enough to find the edge again. See the conclusion for more discussion.

■ Technical Limitations

There are a few direct commands that we chose not to implement in this package. These are concerned with rebooting the NXT brick, changing its firmware, erasing the flash memory, or executing a reset. In essence, these commands destroy the robot.

With the M4N commands and the NXT commands, the robot can be controlled very efficiently and very easily. However, the COM port is not very fast. Crucially, each switch in the direction of communication of the port requires 6 ms. This is the reason why replies should be avoided unless they are absolutely necessary. In some applications, these delays may be so important that the program must be moved to the brick in whole or in part. The PID controller `MotorControl22.rxe` is an example where the program is distributed over the two machines: the computer sends target moves and the brick executes the decisions millisecond by millisecond to achieve the requested move.

■ Conclusion

One recent theory in cognitive science is the embodiment theory. It states that a cognitive system must be in an interactive relation with the world to develop a meaningful representation of the world. This view contradicts the classical artificial intelligence (AI) view in which cognitive agents manipulate tokens or symbols that may not necessarily be related to aspects of the outside world. Hence, according to the AI view, the sensors and actuators are independent of the cognitive functions and may be built separately. This view is tenable as long as we are devising cognitive systems in a virtual world (e.g. using computer-based simulations). However, as soon as physical robots are actually constructed, we see how difficult it is to hold this position.

The simple line-follower program shows this. In a virtual environment, the line-follower simulator can be run at full speed and steering can be made as abrupt as desired, because there is no inertia and no risk that the robot falls over. When the physical robot is built, these become very serious concerns. Of course, the programmer adapts the program to take these risks into account (reducing the speed and lowering the multiplier factor). Likewise, the `white` and `black` readings are calibrated by the programmer in a very simple way. Yet, if the ambient lighting changes, the calibration must be done again. Further, as said earlier, if the multiplier `mult` is too small, the robot may not be able to steer quickly enough around a sharp curve. Finally, the line-follower robot uses wheels. If it used legs instead, flexibility and possible muscle tear would come into play.

All the above concerns can be seen at worst as annoyances, at best as challenges for the programmer. Yet, the simplest animals are immune to all of these concerns. The embodi-

ment theory holds that organisms developed adequate representations by interacting with the world, and that those representations became robust with regard to all the possible sources of fault. Representations and algorithms given by a programmer, on the other hand, may not respond to real-life tests, however simple or elegant they are.

The NXT is a simple yet complete platform to test robots in real life. Combined with *Mathematica*'s computational power, it is easy to develop adaptive algorithms such as neural networks and see what kind of representations are self-constructed by the robot. It remains to be seen what the simplest adaptive line-follower program looks like.

■ Acknowledgments

The author would like to thank Vincent Brault, Dominic Langlois, and Sylvain Chartier from the CONEC laboratory, University of Ottawa, for their help during the development of the *Math4NXT* package.

■ References

- [1] MathWorks. "Set Up a Bluetooth Connection." (Feb 4, 2013) www.mathworks.com/help/simulink/ug/bluetooth-communications.html.
- [2] The LEGO Group. "LEGO MINDSTORMS NXT Bluetooth Developer Kit." (Jan 7, 2013) mindstorms.LEGO.com/en-us/support/files/default.aspx.
- [3] R. Raguet-Schofield. "SerialIO." Wolfram Library Archive. (Jan 7, 2013) library.wolfram.com/infocenter/MathSource/5726.
- [4] J. Sluka. "A PID Controller For LEGO MINDSTORMS Robots." (Jan 7, 2013) www.inpharmix.com/jps/PID_Controller_For_Lego_Mindstorms_Robots.html.
- [5] Institute of Imaging & Computer Vision. "RWTH—MINDSTORMS NXT Toolbox." (Jan 7, 2013) www.mindstorms.rwth-aachen.de/trac/wiki/MotorControl.

D. Cousineau, "Controlling Robots Built with the LEGO® MINDSTORMS® NXT Brick," *The Mathematica Journal*, 2013. [dx.doi.org/doi:10.3888/tmj.15-3](https://doi.org/10.3888/tmj.15-3).

About the Author

Denis Cousineau is a professor at the University of Ottawa in cognitive psychology. He runs research in artificial intelligence as well as on human categorization processes.

Denis Cousineau

École de psychologie
Université d'Ottawa
 136, rue Jean-Jacques Lussier
 Ottawa (ON), K1N 6N5, CANADA
 Denis.Cousineau@UOttawa.ca