Programming Lego Robots using NQC

(Version 3.04, Feb 14, 2002)

by Mark Overmars
(revisions by John Hansen)

Department of Computer Science
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht
the Netherlands

Preface

The Lego MindStorms and CyberMaster robots are wonderful new toys from which a wide variety of robots can be constructed, that can be programmed to do all sorts of complicated tasks. Unfortunately, the software that comes with the robots is, although visually attractive, rather limited in functionality. Hence, it can only be used for simple tasks. To unleash the full power of the robots, you need a different programming environment. NQC is a programming language, written by Dave Baum, which was especially designed for the Lego robots. If you have never written a program before, don't worry. NQC is really easy to use and this tutorial will tell you all about it. Actually, programming the robots in NQC is a lot easier than programming a normal computer, so this is a chance to become a programmer in an easy way.

To make writing programs even easier, there is the Bricx Command Center. This utility helps you to write your programs, to send them to the robot, and to start and stop the robot. Bricx Command Center works almost like a text processor, but with some extras. This tutorial will use Bricx Command Center (version 3.3 or higher) as programming environment. You can download it for free from the web at the address

http://members.aol.com/johnbinder/bricxcc.htm

Bricx Command Center runs on Windows PC's (95, 98, ME, NT, 2K, XP). The language NQC can also be used on other platforms. You can download it from the web at address

http://www.enteract.com/~dbaum/lego/nqc/

Most of this tutorial also applies to the other platforms (assuming you use NQC version 2.0 or higher), except that you loose some of the tools and the color-coding.

In this tutorial I assume that you have the MindStorms (RCX) robot. Most of the contents also applies to the CyberMaster and Scout robots although some of the functionality is not available for those robots. Also the names of e.g. the motors are different so you will have to change the examples a little bit to make them work.

Acknowledgements

I would like to thank Dave Baum for developing NQC. Also many thanks to Kevin Saddi for writing a first version of the first part of this tutorial.

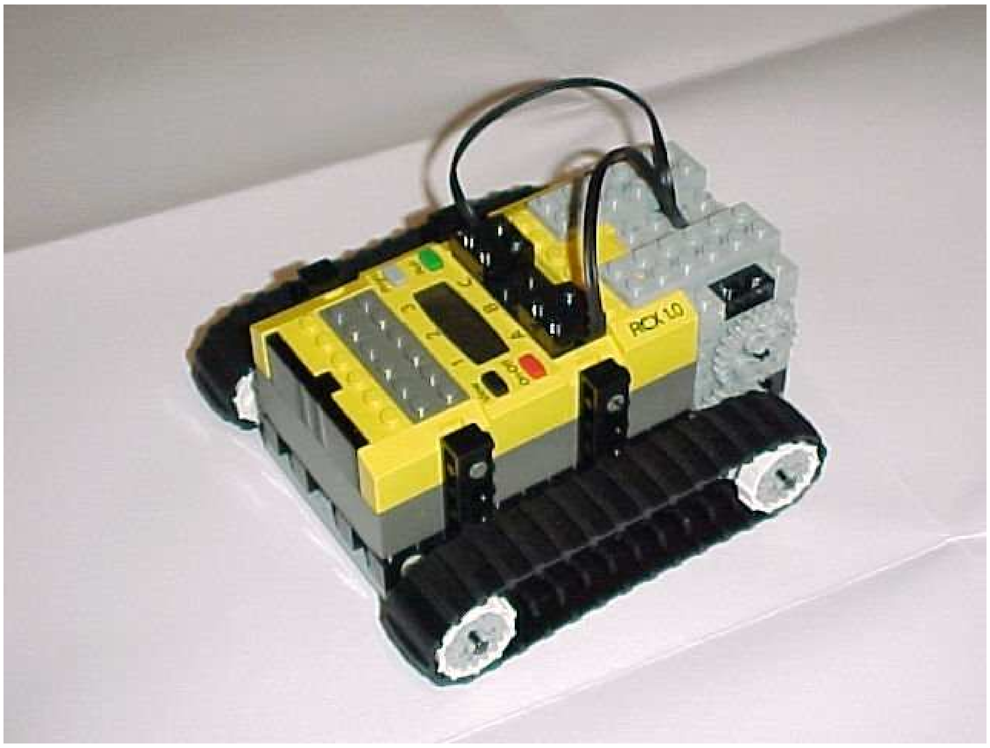
Contents

Writing your first program

In this chapter I will show you how to write an extremely simple program. We are going to program a robot to move forwards for 4 seconds, then backwards for another 4 seconds, and then stop. Not very spectacular but it will introduce you to the basic idea of programming. And it will show you how easy this is. But before we can write a program, we first need a robot.

Building a robot

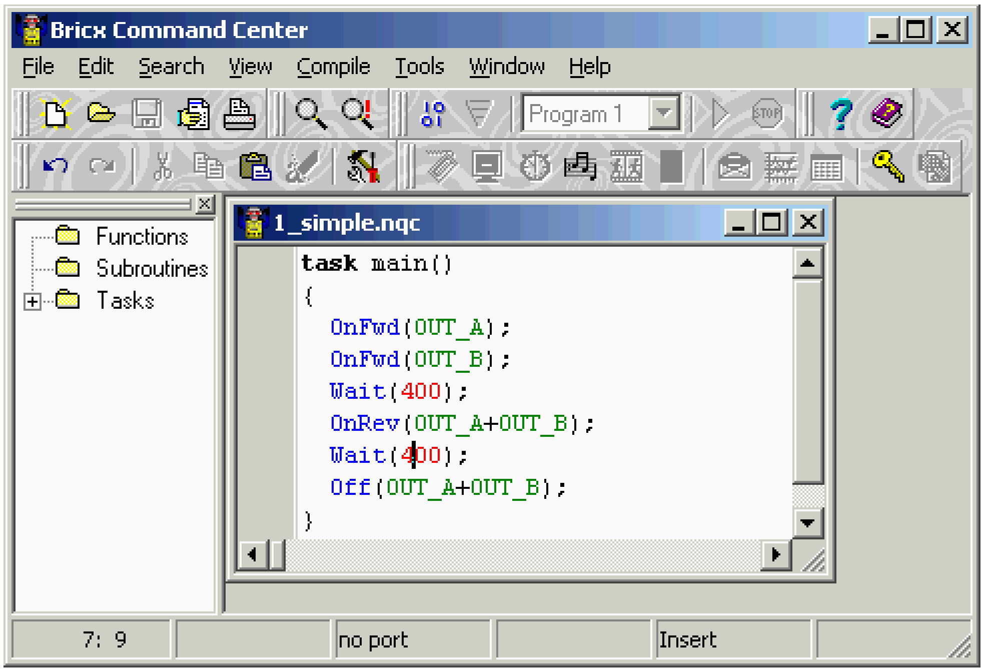
The robot we will use throughout this tutorial is a simple version of the top-secret robot that is described on page 39-46 of your constructopedia. We will only use the basis chassis. Remove the whole front with the two arms and the touch sensors. Also, connect the motors slightly different such that the wires are connected to the RCX at the outside. This is important for your robot to drive in the correct direction. Your robot should look like this:



Also make sure that the infra-red port is correctly connected to your computer and that it is set to long range. (You might want to check with the RIS software that the robot is functioning well.)

Starting Bricx Command Center

We write our programs using Bricx Command Center. Start it by double clicking on the icon BricxCC. (I assume you already installed Bricx Command Center. If not, download it from the web site (see the preface), and install it in any directory you like.) The program will ask you where to locate the robot. Switch the robot on and press **OK**. The program will (most likely) automatically find the robot. Now the user interface appears as shown below (without a window).



The interface looks like a standard text editor, with the usual menu's, and buttons to open and save files, print files, edit files, etc. But there are also some special menus for compiling and downloading programs to the robot and for getting information from the robot. You can ignore these for the moment.

We are going to write a new program. So press the **New File** button to create a new, empty window.

Writing the program

Now type in the following program:

```
task main()
{
  OnFwd(OUT_A);
  OnFwd(OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

It might look a bit complicated at first, so let us analyze it. Programs in NQC consist of tasks. Our program has just one task, named `main`. Each program needs to have a task called `main` which is the one that will be executed by the robot. You will learn more about tasks in Chapter VI. A task consists of a number of commands, also called statements. There are brackets around the statements such that it is clear that they all belong to this task. Each statement ends with a semicolon. In this way it is clear where a statement ends and where the next statement begins. So a task looks in general as follows:

```
task main()
{
  statement1;
  statement2;
    …
}
```

Our program has six statements. Let us look at them one at the time:

`OnFwd(OUT_A);`

This statement tells the robot to start output A, that is, the motor connected to the output labeled A on the RCX, to move forwards. It will move with maximal speed, unless you first set the speed. We will see later how to do this.

`OnFwd(OUT_C);`

Same statement but now we start motor C. After these two statements, both motors are running, and the robot moves forwards.

`Wait(400);`

Now it is time to wait for a while. This statement tells us to wait for 4 seconds. The argument, that is, the number between the parentheses, gives the number of "ticks". Each tick is 1/100 of a second. So you can very precisely tell the program how long to wait. So for 4 seconds, the program does do nothing and the robot continues to move forwards.

`OnRev(OUT_A+OUT_C);`

The robot has now moved far enough so we tell it to move in reverse direction, that is, backwards. Note that we can set both motors at once using `OUT_A+OUT_C` as argument. We could also have combined the first two statements this way.

`Wait(400);`

Again we wait for 4 seconds.

`Off(OUT_A+OUT_C);`

And finally we switch both motors off.

That is the whole program. It moves both motors forwards for 4 seconds, then backwards for 4 seconds, and finally switches them off.

You probably noticed the colors when typing in the program. They appear automatically. The colors and styles used by the editor when it performs syntax highlighting are customizable.

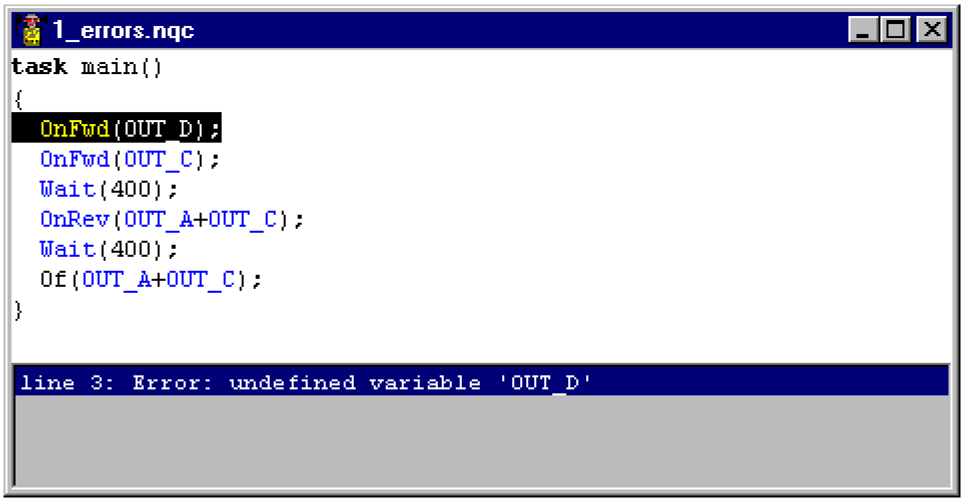
Running the program

Once you have written a program, it needs to be compiled (that is, changed into code that the robot can understand and execute) and send to the robot using the infra red link (called "downloading" the program). There is a button that does both at once (see the figure above). Press this button and, assuming you made no errors when typing in the program, it will correctly compile and be downloaded. (If there are errors in your program you will be notified; see below.)

Now you can run your program. To this end press the green run button on your robot or, more easily, press the run button on your window (see the figure above). Does the robot do what you expected? If not, the wires are probably connected wrong.

Errors in your program

When typing in programs there is a reasonable chance that you make some errors. The compiler notices the errors and reports them to you at the bottom of the window, like in the following figure:



It automatically selects the first error (we mistyped the name of the motor). When there are more errors, you can click on the error messages to go to them. Note that often errors at the beginning of the program cause other errors at other places. So better only correct the first few errors and then compile the program again. Also note that the syntax highlighting helps a lot in avoiding errors. For example, on the last line we typed `Of` rather than `Off`. Because this is an unknown command it is not highlighted.

There are also errors that are not found by the compiler. If we had typed `OUT_B` this would have gone unnoticed because that motor exists (even though we do not use it in the robot). If your robot exhibits unexpected behavior, there is most likely something wrong in your program.

Changing the speed

As you noticed, the robot moved rather fast. Default the robot moves as fast as it can. To change the speed you can use the command `SetPower()`. The power is a number between 0 and 7. 7 is the fastest, 0 the slowest (but the robot will still move). Here is a new version of our program in which the robot moves slow:

```
task main()
{
  SetPower(OUT_A+OUT_C,2);
  OnFwd(OUT_A+OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

Summary

In this chapter you wrote your first program in NQC, using Bricx Command Center. You should now know how to type in a program, how to download it to the robot and how to let the robot execute the program. Bricx Command Center can do many more things. To find out about them, read the documentation that comes with it. This tutorial will primarily deal with the language NQC and only mention features of Bricx Command Center when you really need them.

You also learned some important aspects of the language NQC. First of all, you learned that each program has one task named `main` that is always executed by the robot. Also you learned the four most important motor commands: `OnFwd()`, `OnRev()`, `SetPower()` and `Off()`. Finally, you learned about the `Wait()` statement.

II.A more interesting program

Our first program was not very spectacular. So let us try to make it more interesting. We will do this in a number of steps, introducing some important features of our programming language NQC.

Making turns

You can make your robot turn by stopping or reversing the direction of one of the two motors. Here is an example. Type it in, download it to your robot and let it run. It should drive a bit and then make a 90-degree right turn.

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  OnRev(OUT_C);
  Wait(85);
  Off(OUT_A+OUT_C);
}
```

You might have to try some slightly different numbers than 85 in the second `Wait()` command to make a precise 90-degree turn. This depends on the type of surface on which the robot runs. Rather than changing this in the program it is easier to use a name for this number. In NQC you can define constant values as shown in the following program.

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(MOVE_TIME);
  OnRev(OUT_C);
  Wait(TURN_TIME);
  Off(OUT_A+OUT_C);
}
```

The first two lines define two constants. These can now be used throughout the program. Defining constants is good for two reasons: it makes the program more readable, and it is easier to change the values. Note that Bricx Command Center gives the define statements its own color. As we will see in Chapter VI, you can also define things other than constants.

Repeating commands

Let us now try to write a program that makes the robot drive in a square. Going in a square means: driving forwards, turning 90 degrees, driving forwards again, turning 90 degrees, etc. We could repeat the above piece of code four times but this can be done a lot easier with the **`repeat`** statement.

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  repeat(4)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    OnRev(OUT_C);
    Wait(TURN_TIME);
  }
  Off(OUT_A+OUT_C);
}
```

The number behind the **repeat** statement, between parentheses, indicates how often something must be repeated. The statements that must be repeated are put between brackets, just like the statements in a task. Note that, in the above program, we also indent the statements. This is not necessary, but it makes the program more readable.

As a final example, let us make the robot drive 10 times in a square. Here is the program:

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  repeat(10)
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(MOVE_TIME);
      OnRev(OUT_C);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_A+OUT_C);
}
```

There is now one repeat statement inside the other. We call this a “nested” repeat statement. You can nest repeat statements as much as you like. Take a careful look at the brackets and the indentation used in the program. The task starts at the first bracket and ends at the last. The first repeat statement starts at the second bracket and ends at the fifth. The second, nested repeat statement starts at the third bracket and ends at the fourth. As you see the brackets always come in pairs, and the piece between the brackets we indent.

Adding comment

To make your program even more readable, it is good to add some comment to it. Whenever you put `//` on a line, the rest of that line is ignored and can be used for comments. A long comment can be put between `/*` and `*/`. Comments are syntax highlighted in the Bricx Command Center. The full program could look as follows:

```
/*  10 SQUARES

    by Mark Overmars

This program make the robot run 10 squares
*/

#define MOVE_TIME   100      // Time for a straight move
#define TURN_TIME    85      // Time for turning 90 degrees

task main()
{
  repeat(10)                 // Make 10 squares
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(MOVE_TIME);
      OnRev(OUT_C);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_A+OUT_C);          // Now turn the motors off
}
```

Summary

In this chapter you learned the use of the **`repeat`** statement and the use of comment. Also you saw the function of nested brackets and the use of indentation. With all you know so far you can make the robot move along all sorts of paths. It is a good exercise to try and write some variations of the programs in this chapter before continuing with the next chapter.

III.Using variables

Variables form a very important aspect of every programming language. Variables are memory locations in which we can store a value. We can use that value at different places and we can change it. Let me describe the use of variables using an example.

Moving in a spiral

Assume we want to adapt the above program in such a way that the robot drives in a spiral. This can be achieved by making the time we sleep larger for each next straight movement. That is, we want to increase the value of `MOVE_TIME` each time. But how can we do this? `MOVE_TIME` is a constant and constants cannot be changed. We need a variable instead. Variables can easily be defined in NQC. You can have 32 of these, and you can give each of them a separate name. Here is the spiral program.

```
#define TURN_TIME   85

int move_time;                // define a variable

task main()
{
  move_time = 20;             // set the initial value
  repeat(50)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);         // use the variable for sleeping
    OnRev(OUT_C);
    Wait(TURN_TIME);
    move_time += 5;           // increase the variable
  }
  Off(OUT_A+OUT_C);
}
```

The interesting lines are indicated with the comments. First we define a variable by typing the keyword **`int`** followed by a name we choose. (Normally we use lower-case letters for variable names and uppercase letters for constants, but this is not necessary.) The name must start with a letter but can contain digits and the underscore sign. No other symbols are allowed. (The same applied to constants, task names, etc.) The strange word **`int`** stands for integer. Only integer numbers can be stored in it. In the second interesting line we assign the value 20 to the variable. From this moment on, whenever you use the variable, it stands for 20. Now follows the repeat loop in which we use the variable to indicate the time to sleep and, at the end of the loop we increase the value of the variable with 5. So the first time the robot sleeps 20 ticks, the second time 25, the third time 30, etc.

Besides adding values to a variable we can also multiply a variable with a number using `*=`, subtract using `-=` and divide using `/=`. (Note that for division the result is rounded to the nearest integer.) You can also add one variable to the other, and write down more complicated expressions. Here are some examples:

```
int aaa;
int bbb, ccc;

task main()
{
  aaa = 10;
  bbb = 20 * 5;
  ccc = bbb;
  ccc /= aaa;
  ccc -= 5;
  aaa = 10 * (ccc + 3);  // aaa is now equal to 80
}
```

Note on the first two lines that we can define multiple variables in one line. We could also have combined all three of them in one line.

Random numbers

In all the above programs we defined exactly what the robot was supposed to do. But things get a lot more interesting when the robot is going to do things that we don't know. We want some randomness in the motions. In NQC you can create random numbers. The following program uses this to let the robot drive around in a random way. It constantly drives forwards for a random amount of time and then makes a random turn.

```
int move_time, turn_time;

task main()
{
  while(true)
  {
    move_time = Random(60);
    turn_time = Random(40);
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);
    OnRev(OUT_A);
    Wait(turn_time);
  }
}
```

The program defines two variables, and then assigns random numbers to them. `Random(60)` means a random number between 0 and 60 (it can also be 0 or 60). Each time the numbers will be different. (Note that we could avoid the use of the variables by writing e.g. `Wait(Random(60))`.)

You also see a new type of loop here. Rather that using the repeat statement we wrote **`while(true)`**. The while statement repeats the statements below it as long as the condition between the parentheses is true. The special word **`true`** is always true, so the statements between the brackets are repeated forever, just as we want. You will learn more about the while statement in Chapter IV.

Summary

In this chapter you learned about the use of variables. Variables are very useful but, due to restrictions of the robots, they are a bit limited. You can define only 32 of them and they can store only integers. But for many robot tasks this is good enough.

You also learned how to create random numbers, such that you can give the robot unpredictable behavior. Finally we saw the use of the while statement to make an infinite loop that goes on forever.

IV.Control structures

In the previous chapters we saw the repeat and while statements. These statements control the way the other statements in the program are executed. They are called “control structures”. In this chapter we will see some other control structures.

The if statement

Sometimes you want that a particular part of your program is only executed in certain situations. In this case the if statement is used. Let me give an example. We will again change the program we have been working with so far, but with a new twist. We want the robot to drive along a straight line and then either make a left or a right turn. To do this we need random numbers again. We pick a random number between 0 and 1, that is, it is either 0 or 1. If the number is 0 we make a right turn; otherwise we make a left turn. Here is the program:

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  while(true)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    if (Random(1) == 0)
    {
      OnRev(OUT_C);
    }
    else
    {
      OnRev(OUT_A);
    }
    Wait(TURN_TIME);
  }
}
```

The if statement looks a bit like the while statement. If the condition between the parentheses is true the part between the brackets is executed. Otherwise, the part between the brackets after the word **`else`** is executed. Let us look a bit better at the condition we use. It reads `Random(1) == 0`. This means that `Random(1)` must be equal to 0 to make the condition true. You might wonder why we use == rather than =. The reason is to distinguish it from the statement that put a value in a variable. You can compare values in different ways. Here are the most important ones:

==	equal to
<	smaller than
<=	smaller than or equal to
>	larger than
>=	larger than or equal to
!=	not equal to

You can combine conditions use `&&`, which means “and”, or `||`, which means “or”. Here are some examples of conditions:

`true`	always true
`false`	never true
`ttt != 3`	true when ttt is not equal to 3

`(ttt >= 5) && (ttt <= 10)` true when ttt lies between 5 and 10
`(aaa == 10) || (bbb == 10)` true if either aaa or bbb (or both) are equal to 10

Note that the if statement has two parts. The part immediately after the condition, which is executed when the condition is true, and the part after the else, which is executed when the condition is false. The keyword else and the part after it are optional. So you can leave them away if there is nothing to do when the condition is false.

The do statement

There is another control structure, the do statement. It has the following form:

```
do
{
  statements;
}
while (condition);
```

The statements between the brackets after the do part are executed as long as the condition is true. The condition has the same form as in the if statement described above. Here is an example of a program. The robot runs around randomly for 20 seconds and then stops.

```
int move_time, turn_time, total_time;

task main()
{
  total_time = 0;
  do
  {
    move_time = Random(100);
    turn_time = Random(100);
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);
    OnRev(OUT_C);
    Wait(turn_time);
    total_time += move_time; total_time += turn_time;
  }
  while (total_time < 2000);
  Off(OUT_A+OUT_C);
}
```

Note in this example that we placed two statements on one line. This is allowed. You can place as many statements on a line as you like (as long as there are semicolons in between). But for readability of the program this is often not a good idea.

Note also that the do statement behaves almost the same as the while statement. But in the while statement the condition is tested before executing the statements, while in the do statement the condition is tested at the end. For the while statement, the statements might never be executed, but for the do statement they are executed at least once.

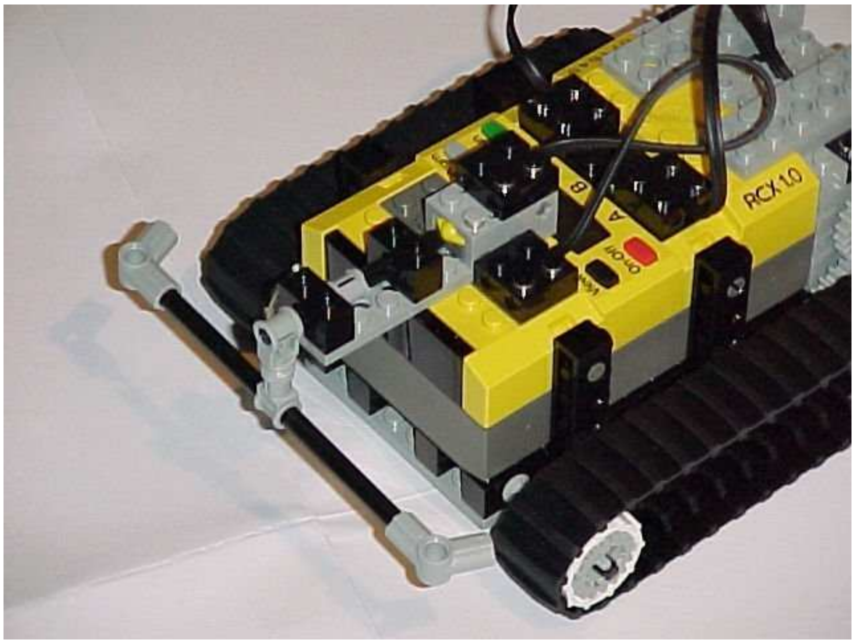
Summary

In this chapter we have seen two new control structures: the if statement and the do statement. Together with the repeat statement and the while statement they are the statements that control the way in which the program is executed. It is very important that you understand what they do. So better try some more examples yourself before continuing.

We also saw that we can place multiple statements on a line.

V.Sensors

One of the nice aspects of the Lego robots is that you can connect sensors to them and that you can make the robot react to the sensors. Before I can show how to do this we must change the robot a bit by adding a sensor. To this end, build the sensor construction shown in figure 4 on page 28 of the constructopedia. You might want to make it slightly wider, such that your robot looks as follows:



Connect the sensor to input 1 on the RCX.

Waiting for a sensor

Let us start with a very simple program in which the robot drives forwards until it hits something. Here it is:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
  until (SENSOR_1 == 1);
  Off(OUT_A+OUT_C);
}
```

There are two important lines here. The first line of the program tells the robot what type of sensor we use. `SENSOR_1` is the number of the input to which we connected the sensor. The other two sensor inputs are called `SENSOR_2` and `SENSOR_3`. `SENSOR_TOUCH` indicates that this is a touch sensor. For the light sensor we would use `SENSOR_LIGHT`. After we specified the type of the sensor, the program switches on both motors and the robot starts moving forwards. The next statement is a very useful construction. It waits until the condition between the brackets is true. This condition says that the value of the sensor `SENSOR_1` must be 1, which means that the sensor is pressed. As long as the sensor is not pressed, the value is 0. So this statement waits until the sensor is pressed. Then we switch off the motors and the task is finished.

Acting on a touch sensor

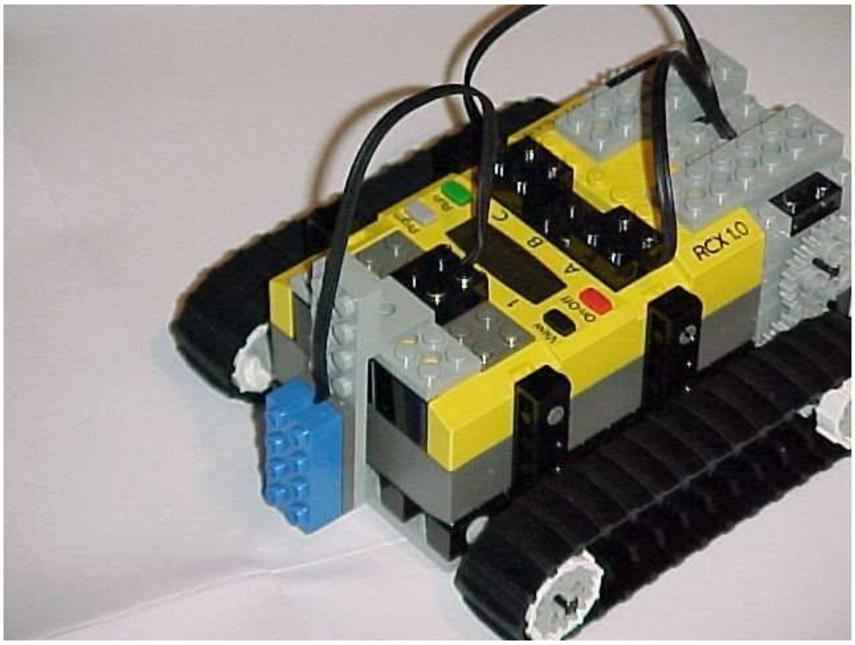
Let us now try to make the robot avoid obstacles. Whenever the robot hits an object, we let it move back a bit, make a turn, and then continue. Here is the program:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_A+OUT_C); Wait(30);
      OnFwd(OUT_A); Wait(30);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

As in the previous example, we first indicate the type of the sensor. Next the robot starts moving forwards. In the infinite while loop we constantly test whether the sensor is touched and, if so, move back for 1/3 of a second, turn right for 1/3 of a second, and then continue forwards again.

Light sensors

Besides touch sensors, you also get a light sensor with your MindStorms system. The light sensor measures the amount of light in a particular direction. The light sensor also emits light. In this way it is possible to point the light sensor in a particular direction and make a distinction between the intensity of the object in that direction. This is in particular useful when trying to make a robot follow a line on the floor. This is what we are going to do in the next example. We first need to attach the light sensor to the robot such that it is in the middle of the robot, at the front, and points downwards. Connect it to input 2. For example, make a construction as follows:



We also need the race track that comes with the RIS kit (This big piece of paper with the black track on it.) The idea now is that the robot makes sure that the light sensor stays above the track. Whenever the intensity of the light goes up, the light sensor is off the track and we need to adapt the direction. Here is a very simple program for this that only works if we travel around the track in clockwise direction.

```
#define THRESHOLD 40

task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_2 > THRESHOLD)
    {
      OnRev(OUT_C);
      until (SENSOR_2 <= THRESHOLD);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

The program first indicates that sensor 2 is a light sensor. Next it sets the robot to move forwards and goes into an infinite loop. Whenever the light value is bigger than 40 (we use a constant here such that this can be adapted easily, because it depends a lot on the surrounding light) we reverse one motor and wait till we are on the track again.

As you will see when you execute the program, the motion is not very smooth. Try adding a `Wait(10)` command before the **until** command to make the robot move better. Note that the program does not work for moving counter-clockwise. To enable motion along arbitrary path a much more complicated program is required.

Summary

In this chapter you have seen how to work with touch sensors and light sensors. We also saw the **until** command that is useful when using sensors.

I recommend you to write a number of programs yourself at his stage. You have all the ingredients to give your robots pretty complicated behavior now. For example, try to put two touch sensors on your robot, one on the left front and the other on the right front, and make the robot move away from the obstacles it hits. Also, try to make a robot that stays within an area indicated by a thick black border line on the floor.

VI.Tasks and subroutines

Up to now all our programs consisted of just one task. But NQC programs can have multiple tasks. It is also possible to put pieces of code in so-called subroutines that you can use at different places in your program. Using tasks and subroutines makes your programs easier to understand and more compact. In this chapter we will look at the various possibilities.

Tasks

An NQC program consists of at most 10 tasks. Each task has a name. One task must have the name `main`, and this task will be executed. The other tasks will only be executed when a running tasks tells them to be executed using a start command. From this moment on both tasks are running simultaneously (so the first task continues running). A running task can also stop another running task by using the stop command. Later this task can be restarted again, but it will start from the beginning; not from the place where it was stopped.

Let me demonstrate the use of tasks. Put your touch sensor again on your robot. We want to make a program in which the robot drives around in squares, like before. But when it hits an obstacle it should react to it. It is difficult to do this in one task, because the robot must do two things at the same moment: drive around (that is, switching on and off motors at the right moments) and watch for sensors. So it is better to use two tasks for this, one task that drives the squares; the other that reacts to the sensors. Here is the program.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start check_sensors;
  start move_square;
}

task move_square()
{
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      stop move_square;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      start move_square;
    }
  }
}
```

The main task just sets the sensor type and then starts both other tasks. After this, task main is finished. Task `move_square` moves the robot forever in squares. Task `check_sensors` checks whether the touch sensor is pushed. If so it takes the following actions: First of all it stops task `move_square`. This is very important. `check_sensors` now takes control over the motions of the robot. Next it moves the robot back a bit and makes it turn. Then it can start `move_square` again to let the robot again drive in squares.

It is very important to remember that tasks that you start are running at the same moment. This can lead to unexpected results. Chapter X explains these problems in detail and gives solutions for them.

Subroutines

Sometimes you need the same piece of code at multiple places in your program. In this case you can put the piece of code in a subroutine and give it a name. Now you can execute this piece of code by simply calling its name from within a task. NQC (or actually the RCX) allows for at most 8 subroutines. Let us look at an example.

```
sub turn_around()
{
  OnRev(OUT_C); Wait(340);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around();
  Wait(200);
  turn_around();
  Wait(100);
  turn_around();
  Off(OUT_A+OUT_C);
}
```

In this program we have defined a subroutine that makes the robot rotate around its center. The main task calls the subroutine three times. Note that we call the subroutine by writing down its name with parentheses behind it. So it looks the same as many of the commands we have seen. Only there are no parameters, so there is nothing between the parentheses.

Some warnings are in place here. Subroutines are a bit weird. For example, subroutines cannot be called from other subroutines. Subroutines can be called from different tasks but this is not encouraged. It very easily leads to problems because the same subroutine might actually be run twice at the same moment by different tasks. This tends to give unwanted effects. Also, when calling a subroutine from different tasks, due to a limitation in the RCX firmware, you cannot use complicated expressions anymore. So, unless you know precisely what you are doing, *don't call a subroutine from different tasks!*

Inline functions

As indicated above, subroutines cause certain problems. The nice part is that they are stored only once in the RCX. This saves memory and, because the RCX does not have so much free memory, this is useful. But when subroutines are short, better use inline functions instead. These are not stored separately but copied at each place they are used. This costs more memory but problems like the ones with using complicated expressions, are no longer present. Also there is no limit on the number of inline functions.

Defining and calling inline functions goes exactly the same way as with subroutines. Only use the keyword **void** rather than **sub**. (The word **void** is used because this same word appears in other languages like C.) So the above example, using inline functions, looks as follows:

```
void turn_around()
{
  OnRev(OUT_C); Wait(340);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around();
  Wait(200);
  turn_around();
  Wait(100);
  turn_around();
  Off(OUT_A+OUT_C);
}
```

Inline functions have another advantage over subroutines. They can have arguments. Arguments can be used to pass a value for certain variables to an inline function. For example, assume, in the above example, we can make the time to turn an argument of the function, as in the following examples:

```
void turn_around(int turntime)
{
  OnRev(OUT_C); Wait(turntime);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around(200);
  Wait(200);
  turn_around(50);
  Wait(100);
  turn_around(300);
  Off(OUT_A+OUT_C);
}
```

Note that in the parenthesis behind the name of the inline function we specify the argument(s) of the function. In this case we indicate that the argument is an integer (there are some other choices) and that its name is turntime. When there are more arguments, you must separate them with commas.

Defining macros

There is yet another way to give small pieces of code a name. You can define macros in NQC (not to be confused with the macros in Bricx Command Center). We have seen before that we can define constants, using #define, by giving them a name. But actually we can define any piece of code. Here is the same program again but now using a macro for turning around.

```
#define turn_around OnRev(OUT_C);Wait(340);OnFwd(OUT_A+OUT_C);

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around;
  Wait(200);
  turn_around;
  Wait(100);
  turn_around;
  Off(OUT_A+OUT_C);
}
```

After the #define statement the word turn_around stands for the text behind it. Now wherever you type turn_around, this is replaced by this text. Note that the text should be on one line. (Actually there are ways of putting a #define statement on multiple lines, but this is not recommended.)

Define statements are actually a lot more powerful. They can also have arguments. For example, we can put the time to turn as an argument in the statement. Here is an example in which we define four macro's; one to move forwards, one to move backwards, one to turn left and one to turn right. Each has two arguments: the speed and the time.

```
#define turn_right(s,t)   SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A);OnRev(OUT_C);Wait(t);
#define turn_left(s,t)    SetPower(OUT_A+OUT_C,s);OnRev(OUT_A);OnFwd(OUT_C);Wait(t);
#define forwards(s,t)     SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A+OUT_C);Wait(t);
#define backwards(s,t)    SetPower(OUT_A+OUT_C,s);OnRev(OUT_A+OUT_C);Wait(t);

task main()
{
  forwards(3,200);
  turn_left(7,85);
  forwards(7,100);
  backwards(7,200);
  forwards(7,100);
  turn_right(7,85);
  forwards(3,200);
  Off(OUT_A+OUT_C);
}
```

It is very useful to define such macros. It makes your code more compact and readable. Also, you can more easily change your code when you e.g. change the connections to the motors.

Summary

In this chapter you saw the use of tasks, subroutines, inline functions, and macros. They have different uses. Tasks normally run at the same moment and take care of different things that have to be done at the same moment. Subroutines are useful when larger pieces of code must be used at different places in the same task. Inline functions are useful when pieces of code must be used a many different places in different tasks, but they use more memory. Finally macros are very useful for small pieces of code that must be used a different places. They can also have parameters, making them even more useful.

Now that you have worked through the chapters up to here, you have all the knowledge you need to make your robot do complicated things. The other chapters in this tutorial teach you about other things that are only important in certain applications.

VII.Making music

The RCX has a built-in speaker that can make sounds and even play simple pieces of music. This is in particular useful when you want to make the RCX tell you that something is happening. But it can also be funny to have the robot make music while it runs around.

Built-in sounds

There are six built-in sounds in the RCX, numbered from 0 to 5. They sound as follows:

0	Key click
1	Beep beep
2	Decreasing frequency sweep
3	Increasing frequency sweep
4	'Buhhh' Error sound
5	Fast increasing sweep

You can play them using the commands `PlaySound()`. Here is a small program that plays all of them.

```
task main()
{
  PlaySound(0); Wait(100);
  PlaySound(1); Wait(100);
  PlaySound(2); Wait(100);
  PlaySound(3); Wait(100);
  PlaySound(4); Wait(100);
  PlaySound(5); Wait(100);
}
```

You might wonder why there are these wait commands. The reason is that the command that plays the sound does not wait for it to finish. It immediately executes the next command. The RCX has a little buffer in which it can store some sounds but after a while this buffer get full and sounds get lost. This is not so serious for sounds but it is very important for music, as we will see below.

Note that the argument to `PlaySound()` must be a constant. You cannot put a variable here!

Playing music

For more interesting music, NQC has the command `PlayTone()`. It has two arguments. The first is the frequency, and the second the duration (in ticks of 1/100h of a second, like in the wait command). Here is a table of useful frequencies:

Sound	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
G#	52	104	208	415	831	1661	3322	
G	49	98	196	392	784	1568	3136	
F#	46	92	185	370	740	1480	2960	
F	44	87	175	349	698	1397	2794	
E	41	82	165	330	659	1319	2637	
D#	39	78	156	311	622	1245	2489	
D	37	73	147	294	587	1175	2349	
C#	35	69	139	277	554	1109	2217	
C	33	65	131	262	523	1047	2093	4186

B	31	62	123	247	494	988	1976	3951
A#	29	58	117	233	466	932	1865	3729
A	28	55	110	220	440	880	1760	3520

As we noted above for sounds, also here the RCX does not wait for the note to finish. So if you use a lot in a row better add (slightly longer) wait commands in between. Here is an example:

```
task main()
{
  PlayTone(262,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(330,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(262,160); Wait(200);
}
```

You can create pieces of music very easily using the Brick Piano that is part of the Bricx Command Center.

If you want to have the RCX play music while driving around, better use a separate task for it. Here you have an example of a rather stupid program where the RCX drives back and forth, constantly making music.

```
task music()
{
  while (true)
  {
    PlayTone(262,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
    PlayTone(330,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
  }
}

task main()
{
  start music;
  while(true)
  {
    OnFwd(OUT_A+OUT_C); Wait(300);
    OnRev(OUT_A+OUT_C); Wait(300);
  }
}
```

Summary

In this chapter you learned how to let the RCX make sounds and music. Also you saw how to use a separate task for music.

VIII. More about motors

There are a number of additional motor commands that you can use to control the motors more precisely. In this chapter we discuss them.

Stopping gently

When you use the `Off()` command, the motor stops immediately, using the brake. In NQC it is also possible to stop the motors in a more gentle way, not using the brake. For this you use the `Float()` command. Sometimes this is better for your robot task. Here is an example. First the robot stops using the brakes; next without using the brakes. Note the difference. (Actually the difference is very small for this particular robot. But it makes a big difference for some other robots.)

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Off(OUT_A+OUT_C);
  Wait(100);
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Float(OUT_A+OUT_C);
}
```

Advanced commands

The command `OnFwd()` actually does two things: it switches the motor on and it sets the direction to forwards. The command `OnRev()` also does two things: it switches the motor on and sets the direction to reverse. NQC also has commands to do these two things separately. If you only want to change one of the two things, it is more efficient to use these separate commands; it uses less memory in the RCX, it is faster, and it can result in smoother motions. The two separate commands are `SetDirection()` that sets the direction (`OUT_FWD`, `OUT_REV` or `OUT_TOGGLE` which flips the current direction) and `SetOutput()` that sets the mode (`OUT_ON`, `OUT_OFF` or `OUT_FLOAT`). Here is a simple program that makes the robot drive forwards, backwards and forwards again.

```
task main()
{
  SetPower(OUT_A+OUT_C,7);
  SetDirection(OUT_A+OUT_C,OUT_FWD);
  SetOutput(OUT_A+OUT_C,OUT_ON);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_REV);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_TOGGLE);
  Wait(200);
  SetOutput(OUT_A+OUT_C,OUT_FLOAT);
}
```

Note that, at the start of every program, all motors are set in forward direction and the speed is set to 7. So in the above example, the first two commands are not necessary.

There are a number of other motor commands, which are shortcuts for combinations of the commands above. Here is a complete list:

`On('motors')`	Switches the motors on
`Off('motors')`	Switches the motors off

`Float('motors')`	Switches the motors of smoothly
`Fwd('motors')`	Switches the motors forward (but does not make them drive)
`Rev('motors')`	Switches the motors backwards (but does not make them drive)
`Toggle('motors')`	Toggles the direction of the motors (forward to backwards and back)
`OnFwd('motors')`	Switches the motors forward and turns them on
`OnRev('motors')`	Switches the motors backwards and turns them on
`OnFor('motors','ticks')`	Switches the motors on for ticks time
`SetOutput('motors','mode')`	Sets the output mode (`OUT_ON`, `OUT_OFF` or `OUT_FLOAT`)
`SetDirection('motors','dir')`	Sets the output direction (`OUT_FWD`, `OUT_REV` or `OUT_TOGGLE`)
`SetPower('motors','power')`	Sets the output power (0-9)

Varying motor speed

As you probably noticed, changing the speed of the motors does not have much effect. The reason is that you are mainly changing the torque, not the speed. You will only see an effect when the motor has a heavy load. And even then, the difference between 2 and 7 is very small. If you want to have better effects the trick is to turn the motors on and off in rapid succession. Here is a simple program that does this. It has one task, called run_motor that drives the motors. It constantly checks the variable speed to see what the current speed is. Positive is forwards, negative backwards. It sets the motors in the right direction and then waits for some time, depending on speed, before switching the motors off again. The main task simply sets speeds and waits.

```
int speed, __speed;

task run_motor()
{
  while (true)
  {
    __speed = speed;
    if (__speed > 0) {OnFwd(OUT_A+OUT_C);}
    if (__speed < 0) {OnRev(OUT_A+OUT_C); __speed = -__speed
;}
    Wait(__speed);
    Off(OUT_A+OUT_C);
  }
}

task main()
{
  speed = 0;
  start run_motor;
  speed = 1;   Wait(200);
  speed = -10; Wait(200);
  speed = 5;   Wait(200);
  speed = -2;  Wait(200);
  stop run_motor;
  Off(OUT_A+OUT_C);
}
```

This program can be made much more powerful, allowing for rotations, and also possibly incorporating a waiting time after the `Off()` command. Experiment yourself.

Summary

In this chapter you learned about the extra motor commands that are available: `Float()` that stops the motor gently, `SetDirection()` that sets the direction (`OUT_FWD`, `OUT_REV` or `OUT_TOGGLE` which flips the current direction) and `SetOutput()` that sets the mode (`OUT_ON`, `OUT_OFF` or `OUT_FLOAT`). You saw the complete list of motor commands available. You also learned a trick to control the motor speed in a better way.

IX.More about sensors

In Chapter V we discussed the basic aspects of using sensors. But there is a lot more you can do with sensors. In this chapter we will discuss the difference between sensor mode and sensor type, we will see how to use the rotation sensor (a type of sensor that is not provided with the RIS but can be bought separately and is very useful), and we will see some tricks to use more than three sensors and to make a proximity sensor.

Sensor mode and type

The `SetSensor()` command that we saw before does actually two things: it sets the type of the sensor, and it sets the mode in which the sensor operates. By setting the mode and type of the a sensor separately, you can control the behavior of the sensor more precisely, which is useful for particular applications.

The type of the sensor is set with the command `SetSensorType()`. There are four different types: `SENSOR_TYPE_TOUCH`, which is the touch sensor, `SENSOR_TYPE_LIGHT`, which is the light sensor, `SENSOR_TYPE_TEMPERATURE`, which is the temperature sensor (this type of sensor is not part of the RIS but can be bought separately), and `SENSOR_TYPE_ROTATION`, which is the rotation sensor (also not part of the RIS but available separately). Setting the type sensor is in particular important to indicate whether the sensor needs power (like e.g. for the light of the light sensor). I know of no uses for setting a sensor to a different type than it actually is.

The mode of the sensor is set with the command `SetSensorMode()`. There are eight different modes. The most important one is `SENSOR_MODE_RAW`. In this mode, the value you get when checking the sensor is a number between 0 and 1023. It is the raw value produced by the sensor. What it means depends on the actual sensor. For example, for a touch sensor, when the sensor is not pushed the value is close to 1023. When it is fully pushed, it is close to 50. When it is pushed partially the value ranges between 50 and 1000. So if you set a touch sensor to raw mode you can actually find out whether it is touched partially. When the sensor is a light sensor, the value ranges from about 300 (very light) to 800 (very dark). This gives a much more precise value than using the `SetSensor()` command.

The second sensor mode is `SENSOR_MODE_BOOL`. In this mode the value is 0 or 1. When the raw value is above about 550 the value is 0, otherwise it is 1. `SENSOR_MODE_BOOL` is the default mode for a touch sensor. The modes `SENSOR_MODE_CELSIUS` and `SENSOR_MODE_FAHRENHEIT` are useful with temperature sensors only and give the temperature in the indicated way. `SENSOR_MODE_PERCENT` turns the raw value into a value between 0 and 100. Every raw value of 400 or lower is mapped to 100 percent. If the raw value gets higher, the percentage slowly goes down to 0. `SENSOR_MODE_PERCENT` is the default mode for a light sensor. `SENSOR_MODE_ROTATION` seems to be useful only for the rotation sensor (see below).

There are two other interesting modes: `SENSOR_MODE_EDGE` and `SENSOR_MODE_PULSE`. They count transitions, that is changes from a low to a high raw value or opposite. For example, when you touch a touch sensor this causes a transition from high to low raw value. When you release it you get a transition the other direction. When you set the sensor mode to `SENSOR_MODE_PULSE`, only transitions from low to high are counted. So each touch and release of the touch sensor counts for one. When you set the sensor mode to `SENSOR_MODE_EDGE`, both transitions are counted. So each touch and release of the touch sensor counts for two. So you can use this to count how often a touch sensor is pushed. Or you can use it in combination with a light sensor to count how often a (strong) lamp is switched on and off. Of course, when you are counting things, you should be able to set the counter back to 0. For this you use the command `ClearSensor()`. It clears the counter for the indicated sensor(s).

Let us look at an example. The following program uses a touch sensor to steer the robot. Connect the touch sensor with a long wire to input one. If touch the sensor quickly twice the robot moves forwards. It you touch it once it stops moving.

```
task main()
{
  SetSensorType(SENSOR_1,SENSOR_TYPE_TOUCH);
  SetSensorMode(SENSOR_1,SENSOR_MODE_PULSE);
  while(true)
  {
    ClearSensor(SENSOR_1);
    until (SENSOR_1 >0);
    Wait(100);
    if (SENSOR_1 == 1) {Off(OUT_A+OUT_C);}
    if (SENSOR_1 == 2) {OnFwd(OUT_A+OUT_C);}
  }
}
```

Note that we first set the type of the sensor and then the mode. It seems that this is essential because changing the type also effects the mode.

The rotation sensor

The rotation sensor is a very useful type of sensor that is unfortunately not part of the standard RIS. It can though be bought separately from Lego. The rotation sensor contains a hole through which you can put an axle. The rotation sensor measures the amount the axle is rotated. One full rotation of the axle is 16 steps (or –16 if you rotate it the other way). Rotation sensors are very useful to make the robot make precisely controlled movements. You can make an axle move the exact amount you want. If you need finer control than 16 step, you can always use gears to connect it to an axle that moves faster, and use that one for counting steps.

One standard application is to have two rotation sensors connected to the two wheels of the robot that you control with the two motors. For a straight movement you want both wheels to turn equally fast. Unfortunately, the motors normally don't run at exactly the same speed. Using the rotation sensors you can see that one wheel turns faster. You can then temporarily stop that motor (best using `Float()`) until both sensors give the same value again. The following program does this. It simply lets the robot drive in a straight line. To use it, change your robot by connecting the two rotation sensors to the two wheels. Connect the sensors to input 1 and 3.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_ROTATION); ClearSensor(SENSOR_1);
  SetSensor(SENSOR_3,SENSOR_ROTATION); ClearSensor(SENSOR_3);
  while (true)
  {
    if (SENSOR_1 < SENSOR_3)
      {OnFwd(OUT_A); Float(OUT_C);}
    else if (SENSOR_1 > SENSOR_3)
      {OnFwd(OUT_C); Float(OUT_A);}
    else
      {OnFwd(OUT_A+OUT_C);}
  }
}
```

The program first indicates that both sensors are rotation sensors, and resets the values to zero. Next it start an infinite loop. In the loop we check whether the two sensor readings are equal. If they are the robot simply moves forwards. If one is larger, the correct motor is stopped until both readings are again equal.

Clearly this is only a very simple program. You can extend this to make the robot drive exact distances, or to let it make very precise turns.

Putting multiple sensors on one input

The RCX has only three inputs so you can connect only three sensors to it. When you want to make more complicated robots (and you bought some extra sensors) this might not be enough for you. Fortunately, with some tricks, you can connect two (or even more) sensors to one input.

The easiest is to connect two touch sensors to one input. If one of them (or both) is touched, the value is 1, otherwise it is 0. You cannot distinguish the two but sometimes this is not necessary. For example, when you put one touch sensor at the front and one at the back of the robot, you know which one is touched based on the direction the robot is driving in. But you can also set the mode of the input to raw (see above). Now you can get a lot more information. If you are lucky, the value when the sensor is pressed is not the same for both sensors. If this is the case you can actually distinguish between the two sensors. And when both are pressed you get a much lower value (around 30) so you can also detect this.

You can also connect a touch sensor and a light sensor to one input. Set the type to light (otherwise the light sensor won't work). Set the mode to raw. In this case, when the touch sensor is pushed you get a raw value below 100. If it is not pushed you get the value of the light sensor which is never below 100. The following program uses this idea. The robot must be equipped with a light sensor pointing down, and a bumper at the front connected to a touch sensor. Connect both of them to input 1. The robot will drive around randomly within a light area. When the light sensor sees a dark line (raw value > 750) it goes back a bit. When the touch sensor touches something (raw value below 100) it does the same. Here is the program:

```
int ttt,tt2;

task moverandom()
{
  while (true)
  {
    ttt = Random(50) + 40;
    tt2 = Random(1);
    if (tt2 > 0)
      { OnRev(OUT_A); OnFwd(OUT_C); Wait(ttt); }
    else
      { OnRev(OUT_C); OnFwd(OUT_A);Wait(ttt); }
    ttt = Random(150) + 50;
    OnFwd(OUT_A+OUT_C);Wait(ttt);
  }
}

task main()
{
  start moverandom;
  SetSensorType(SENSOR_1,SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_1,SENSOR_MODE_RAW);
  while (true)
  {
    if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
    {
      stop moverandom;
      OnRev(OUT_A+OUT_C);Wait(30);
      start moverandom;
    }
  }
}
```

I hope the program is clear. There are two tasks. Task `moverandom` makes the robot move around in a random way. The main task first starts `moverandom`, sets the sensor and then waits for something to happen. If the sensor reading gets too low (touching) or too high (out of the white area) it stops the random moves, backs up a little, and start the random moves again.

It is also possible to connect two light sensors to the same input. The raw value is in some way related to the combined amount of light received by the two sensors. But this is rather unclear and seems hard to use. Connecting other sensors with rotation or temperature sensors seems not to be useful.

Making a proximity sensor

Using touch sensors, your robot can react when it hits something. But it would be a lot nicer when the robot could react just before it hits something. It should know that it is near to some obstacle. Unfortunately there are no sensors for this available. There is though a trick we can use for this. The robot has an infra-red port with which it can communicate with the computer, or with other robots. (We will see more about the communication between robots in Chapter XI.) It turns out that the light sensor that comes with the robot is very sensitive to infra-red light. We can build a proximity sensor based on this. The idea is that one tasks sends out infra-red messages. Another task measures fluctuations in the light intensity that is reflected from objects. The higher the fluctuation, the closer we are to an object.

To use this idea, place the light sensor above the infra-red port on the robot, pointing forwards. In this way it only measures reflected infra-red light. Connect it to input 2. We use raw mode for the light sensor to see the fluctuations as good as possible. Here is a simple program that lets the robot run forwards until it gets near to an object and then makes a 90 degree turn to the right.

```
int lastlevel;            // To store the previous level

task send_signal()
{
  while(true)
    {SendMessage(0); Wait(10);}
}

task check_signal()
{
  while(true)
  {
    lastlevel = SENSOR_2;
    if(SENSOR_2 > lastlevel + 200)
      {OnRev(OUT_C); Wait(85); OnFwd(OUT_A+OUT_C);}
  }
}

task main()
{
  SetSensorType(SENSOR_2, SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_2, SENSOR_MODE_RAW);
  OnFwd(OUT_A+OUT_C);
  start send_signal;
  start check_signal;
}
```

The task `send_signal` send out 10 IR signals every seconds, using the command `SendMessage(0)`. The task `check_signal` repeatedly saves the value of the light sensor. Then it checks whether it (slightly later) has become at least 200 higher, indicating a large fluctuation. If so, it lets the robot make a 90-degree turn to the right. The value of 200 is rather arbitrary. If you make it smaller, the robot turns further away from obstacles. If you make it larger, it gets closer to them. But this also depends on the type of material and the amount of light available in the room. You should experiment or use some more clever mechanism for learning the correct value.

A disadvantage of the technique is that it only works in one direction. You probably still need touch sensors at the sides to avoid collisions there. But the technique is very useful for robots that must drive around in mazes. Another disadvantage is that you cannot communicate from the computer to the robot because it will interfere with the infra-red commands send out by the robot. (Also the remote control on your television might not work.)

Summary

In this chapter we have seen a number of additional issues about sensors. We saw how to separately set the type and mode of a sensor and how this could be used to get additions information. We learned how to use the rotation sensor. And we saw how multiple sensors can be connected to one input of the RCX. Finally, we saw a trick to use the infra red connection of the robot in combination with a light sensor, to create a proximity sensor. All these tricks are extremely useful when constructing more complicated robots. Sensors always play a crucial role there.

X.Parallel tasks

As has been indicated before, tasks in NQC are executed simultaneously, or in parallel as people usually say. This is extremely useful. In enables you to watch sensors in one task while another task moves the robot around, and yet another task plays some music. But parallel tasks can also cause problems. One task can interfere with another.

A wrong program

Consider the following program. Here one task drives the robot around in squares (like we did so often before) and the second task checks for the touch sensor. When the sensor is touched, it moves a bit backwards, and makes a 90-degree turn.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start check_sensors;
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_A+OUT_C);
      Wait(50);
      OnFwd(OUT_A);
      Wait(85);
      OnFwd(OUT_C);
    }
  }
}
```

This probably looks like a perfectly valid program. But if you execute it you will most likely find some unexpected behavior. Try the following: Make the robot touch something while it is turning. It will start going back, but immediately moves forwards again, hitting the obstacle. The reason for this is that the tasks may interfere. The following is happening. The robot is turning right, that is, the first task is in its second sleep statement. Now the robot hits the sensor. It start going backwards, but at that very moment, the main task is ready with sleeping and moves the robot forwards again; into the obstacle. The second task is sleeping at this moment so it won't notice the collision. This is clearly not the behavior we would like to see. The problem is that, while the second task is sleeping we did not realize that the first task was still running, and that its actions interfere with the actions of the second task.

Stopping and restarting tasks

One way of solving this problem is to make sure that at any moment only one task is driving the robot. This was the approach we took in Chapter VI. Let me repeat the program here.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start check_sensors;
  start move_square;
}

task move_square()
{
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      stop move_square;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      start move_square;
    }
  }
}
```

The crux is that the `check_sensors` task only moves the robot after stopping the `move_square` task. So this task cannot interfere with the moving away from the obstacle. Once the backup procedure is finished, it starts `move_square` again.

Even though this is a good solution for the above problem, there is a problem. When we restart `move_square`, it starts again at the beginning. This is fine for our small task, but often this is not the required behavior. We would prefer to stop the task where it is and continue it later from that point. Unfortunately this cannot be done easily.

Using semaphores

A standard technique to solve this problem is to use a variable to indicate which task is in control of the motors. The other tasks are not allowed to drive the motors until the first task indicates, using the variable, that it is ready. Such a variable is often called a semaphore. Let `sem` be such a semaphore. We assume that a value of 0 indicates that no task is steering the motors. Now, whenever a task wants to do something with the motors it executes the following commands:

```
until (sem == 0);
sem = 1;
// Do something with the motors
sem = 0;
```

So we first wait till nobody needs the motors. Then we claim the control by setting `sem` to 1. Now we can control the motors. When we are done we set `sem` back to 0. Here you find the program above, implemented using a semaphore. When the touch sensor touches something, the semaphore is set and the backup procedure is performed. During this procedure the task `move_square` must wait. At the moment the back-up is ready, the semaphore is set to 0 and `move_square` can continue.

```
int sem;

task main()
{
  sem = 0;
  start move_square;
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      until (sem == 0); sem = 1;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      sem = 0;
    }
  }
}

task move_square()
{
  while (true)
  {
    until (sem == 0); sem = 1;
    OnFwd(OUT_A+OUT_C);
    sem = 0;
    Wait(100);
    until (sem == 0); sem = 1;
    OnRev(OUT_C);
    sem = 0;
    Wait(85);
  }
}
```

You could argue that it is not necessary in `move_square` to set the semaphore to 1 and back to 0. Still this is useful. The reason is that the `OnFwd()` command is in fact two commands (see Chapter VIII). You don't want this command sequence to be interrupted by the other task.

Semaphores are very useful and, when you are writing complicated programs with parallel tasks, they are almost always required. (There is still a slight chance they might fail. Try to figure out why.)

Summary

In this chapter we studied some of the problems that can occur when you use different tasks. Always be very careful for side effects. Much unexpected behavior is due to this. We saw two different ways of solving such problems. The first solution stops and restarts tasks to make sure that only one critical task is running at every moment. The second approach uses semaphores to control the execution of tasks. This guarantees that at every moment only the critical part of one task is executed.

XI.Communication between robots

If you own more than one RCX this chapter is for you. The robots can communicate with each other through the infra-red port. Using this you can have multiple robots collaborate (or fight with each other). Also you can build one big robot using two RCXs, such that you can have six motors and six sensors (or even more using the tricks in Chapter IX).

Communication between robots works, globally speaking, as follows. A robot can use the command `SendMessage()` to send a value (0-255) over the infra-red port. All other robots receive this message and store it. The program in a robot can ask for the value of the last message received using `Message()`. Based on this value the program can make the robot perform certain actions.

Giving orders

Often, when you have two or more robots, one is the leader. We call him the *master*. The other robots are *slaves*. The master robot sends orders to the slaves and the slaves execute these. Sometimes the slaves might send information back to the master, for example the value of a sensor. So you need to write two programs, one for the master and one for the slave(s). From now on we assume that we have just one slave. Let us start with a very simple example. Here the slave can perform three different orders: move forwards, move backwards, and stop. Its program consists of a simple loop. In this loop it sets the value of the current message to 0 using the `ClearMessage()` command. Next it waits until the message becomes unequal to 0. Based on the value of the message it executes one of the three orders. Here is the program.

```
task main()            // SLAVE
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

The master has an even simpler program. It simply send the messages corresponding to orders and then waits a bit. In the program below it orders the slave to move forwards, then, after two seconds, backwards, and then, again after two seconds, to stop.

```
task main()            // MASTER
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}
```

After you have written these two program, you need to download them to the robots. Each program must go to one of the robots. Make sure you switch the other one off in the meantime (see also the cautions below). Now switch on both robots and start the programs: first the one in the slave and then the one in the master.

If you have multiple slaves, you have to download the slave program to each of them in turn (not simultaneously; see below). Now all slaves will perform exactly the same actions.

To let the robots communicate with each other we defined, what is called, a protocol: We decided that a 1 means to move forwards, a 2 to move backwards, and a 3 to stop. It is very important to carefully define such protocols, in particular when you are dealing with lots of communications. For example, when there are more slaves, you could define a protocol in which two numbers are sent (with a small sleep in between): the first number is the number of the slave, and the second is the actual order. The slave than first check the number and only perform the action if it is his number. (This requires that each slave has its own number, which can be achieved by letting each slave have a slightly different program in which e.g. one constant is different.)

Electing a leader

As we saw above, when dealing with multiple robots, each robot must have its own program. It would be much easier if we could download just one program to all robots. But then the question is: who is the master? The answer is easy: let the robots decide themselves. Let them elect a leader which the others will follow. But how do we do this? The idea is rather simple. We let each robot wait a random amount of time and then send a message. The one that sends a message first is the leader. This scheme might fail if two robots wait exactly the same amount of time but this is rather unlikely. (You can build more complicated schemes that detect this and try a second election in such a case.) Here is the program that does it:

```
task main()
{
  ClearMessage();
  Wait(200);               // make sure all robots are on
  Wait(Random(400));        // wait between 0 and 4 seconds
  if (Message() > 0)        // somebody else was first
  {
    start slave;
  }
  else
  {
    SendMessage(1);         // I am the master now
    Wait(400);             // make sure everybody else knows
    start master;
  }
}

task master()
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}

task slave()
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

Download this program to all robots (one by one, not at the same moment; see below). Start the robots at about the same moment and see what happens. One of them should take command and the other(s) should follow the orders. In rare occasions, none of them becomes the leader. As indicated above, this requires more careful protocols to solve.

Cautions

You have to be a bit careful when dealing with multiple robots. There are two problems: If two robots (or a robot and the computer) send information at the same time this might be lost. The second problem is that, when the computer sends a program to multiple robots at the same time, this causes problems.

Let us start with the second problem. When you download a program to the robot, the robot tells the computer whether it correctly receives (parts of) the program. The computer reacts on that by sending new pieces or by resending parts. When two robots are on, both will start telling the computer whether they correctly receive the program. The computer does not understand this (it does not know that there are two robots!). As a result, things go wrong and the program gets corrupted. The robots won't do the right things. *Always make sure that, while you are downloading programs, only one robot is on!*

The other problem is that only one robot can send a message at any moment. If two messages are being send at roughly the same moment, they might get lost. Also, a robot cannot send and receive messages at the same moment. This is no problem when only one robot sends messages (there is only one master) but otherwise it might be a serious problem. For example, you can imagine writing a program in which a slave sends a message when it bumps into something, such that the master can take action. But if the master sends an order at the same moment, the message will get lost. To solve this, it is important to define your communication protocol such that, in case a communication fails, this is corrected. For example, when the master sends a command, it should get an answer from the slave. If is does not get an answer soon enough, it resends the command. This would result in a piece of code that looks like this:

```
do
{
  SendMessage(1);
  ClearMessage();
  Wait(10);
}
while (Message() != 255);
```

Here 255 is used for the acknowledgement.

Sometimes, when you are dealing with multiple robots, you might want that only a robot that is very close by receives the signal. This can be achieved by adding the command `SetTxPower(TX_POWER_LO)` to the program of the master. In this case the IR signal send is very low and only a robot close by and facing the master will "hear" it. This is in particular useful when building one bigger robot out of two RCXs. Use `SetTxPower(TX_POWER_HI)` to set the robot again in long range transmission mode.

Summary

In this chapter we studied some of the basic aspects of communication between robots. Communication uses the commands to send, clear, and check messages. We saw that is is important to define a protocol for how the communication works. Such protocols play a crucial role in any form of communication between computers. We also saw that there are a number of restrictions in the communication between robots which makes it even more important to define good protocols.

XII.More commands

NQC has a number of additional commands. In this chapter we will discuss three types: the use of timers, commands to control the display, and the use of the datalog feature of the RCX.

Timers

The RCX has four built-in timers. These timers tick in increments of 1/10 of a second. The timers are numbered from 0 to 3. You can reset the value of a timer with the command `ClearTimer()` and get the current value of the timer with `Timer()`. Here is an example of the use of a timer. The following program lets the robot drive sort of random for 20 seconds.

```
task main()
{
  ClearTimer(0);
  do
  {
    OnFwd(OUT_A+OUT_C);
    Wait(Random(100));
    OnRev(OUT_C);
    Wait(Random(100));
  }
  while (Timer(0)<200);
  Off(OUT_A+OUT_C);
}
```

You might want to compare this program with the one given in Chapter IV that did exactly the same task. The one with timers is definitely simpler.

Timers are very useful as a replacement for a `Wait()` command. You can sleep for a particular amount of time by resetting a timer and then waiting till it reaches a particular value. But you can also react on other events (e.g. from sensors) while waiting. The following simple program is an example of this. It lets the robot drive until either 10 seconds are past, or the touch sensor touches something.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  ClearTimer(3);
  OnFwd(OUT_A+OUT_C);
  until ((SENSOR_1 == 1) || (Timer(3) >100));
  Off(OUT_A+OUT_C);
}
```

Don't forget that timers work in ticks of 1/10 of a second, while e.g. the wait command uses ticks of 1/100 of a second.

The display

It is possible to control the display of the RCX in two different ways. First of all, you can indicate what to display: the system clock, one of the sensors, or one the motors. This is equivalent to using the black view button on the RCX. To set the display type, use the command `SelectDisplay()`. The following program shows all seven possibilities, one after the other.

```
task main()
{
  SelectDisplay(DISPLAY_SENSOR_1); Wait(100);  // Input 1
  SelectDisplay(DISPLAY_SENSOR_2); Wait(100);  // Input 2
  SelectDisplay(DISPLAY_SENSOR_3); Wait(100);  // Input 3
  SelectDisplay(DISPLAY_OUT_A);    Wait(100);  // Output A
  SelectDisplay(DISPLAY_OUT_B);    Wait(100);  // Output B
  SelectDisplay(DISPLAY_OUT_C);    Wait(100);  // Output C
  SelectDisplay(DISPLAY_WATCH);    Wait(100);  // System
clock
}
```

Note that you should not use `SelectDisplay(SENSOR_1)`.

The second way you can control the display is by controlling the value of the system clock. You can use this to display e.g. diagnostic information. For this use the command `SetWatch()`. Here is a tiny program that uses this:

```
task main()
{
  SetWatch(1,1); Wait(100);
  SetWatch(2,4); Wait(100);
  SetWatch(3,9); Wait(100);
  SetWatch(4,16); Wait(100);
  SetWatch(5,25); Wait(100);
}
```

Note that the arguments to `SetWatch()` must be constants.

Datalogging

The RCX can store values of variables, sensor readings, and timers, in a piece of memory called the datalog. The values in the datalog cannot be used inside the RCX, but they can be read by your computer. This is useful to e.g. check what is going on in your robot. Bricx Command Center has a special window in which you can view the current contents of the datalog.

Using the datalog consists of three steps: First, the NQC program must define the size of the datalog, using the command `CreateDatalog()`. This also clears the current contents of the datalog. Next, values can be written in the datalog using the command `AddToDatalog()`. The values will be written one after the other. (If you look at the display of the RCX you will see that one after the other, four parts of a disk appear. When the disk is complete, the datalog is full.) If the end of the datalog is reached, nothing happens. New values are no longer stored. The third step is to upload the datalog to the PC. For this, choose in Bricx Command Center the command **Datalog** in the **Tools** menu. Next press the button labelled **Upload Datalog**, and all the values appear. You can watch them or save them to a file to do something else with them. People have used this feature to e.g. make a scanner with the RCX.

Here is a simple example of a robot with a light sensor. The robot drives for 10 seconds, and five times a second the value of the light sensor is written into the datalog.

```
task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  CreateDatalog(50);
  repeat (50)
  {
    AddToDatalog(SENSOR_2);
    Wait(20);
  }
  Off(OUT_A+OUT_C);
}
```

XIII. Final remarks

If you have worked your way through this tutorial you can now consider yourself an expert in NQC. If you have not done this up to now, it is time to start experimenting yourself. With creativity in design and programming you can make Lego robots do the most wonderful things.

This tutorial did not cover all aspects of the Bricx Command Center. You are recommended to read the documentation at some stage. Also NQC is still in development. Future version might incorporate additional functionality. Many programming concepts were not treated in this tutorial. In particular, we did not consider learning behavior of robots or other aspects of artificial intelligence.

It is also possible to steer a Lego robot directly from a PC. This requires you to write a program in a language like Visual Basic, Java or Delphi. It is also possible to let such a program work together with an NQC program running in the RCX itself. Such a combination is very powerful. If you are interested in this way of programming your robot, best start with downloading the spirit technical reference from the Lego MindStorms web site.

> http://www.legomindstorms.com/

The web is a perfect source for additional information. Some other important starting points are on LUGNET, the LEGO® Users Group Network (unofficial):

> http://www.lugnet.com/

A lot of information can also be found in the newsgroup lugnet.robotics and lugnet.robotics.rcx.nqc at news.lugnet.com.