Programming Lego Robots using NBC

(Version 1.0, June 11, 2007)

(Requires NBC 1.0.1.b30 or greater)

by Ross Crawford

with revisions by John Hansen

Preface

The Lego MindStorms NXT robot is a wonderful new toy 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 robot, although visually attractive and much more powerful than the RIS software for the RCX, is still somewhat limited in its functionality. To unleash the full power of your robot, you need a different programming environment. NBC is a programming language, written by John Hansen, which is especially designed for the Lego robots. If you have never written a program before, don't worry. NBC is really easy to use and this tutorial will tell you all about it. Actually, programming robots in NBC 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 3.3.7.15 or higher) as programming environment. You can download it for free from the web at the address

http://bricxcc.sourceforge.net/

Bricx Command Center runs on Windows PC's (95, 98, ME, NT, 2K, XP). The language NBC can also be downloaded from the web at address

http://bricxcc.sourceforge.net/nbc/

Acknowledgements

I would like to thank John Hansen for developing NBC. Also many thanks to Mark Overmars for writing his NQC tutorial, on which this is heavily based.

Contents

I. 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.

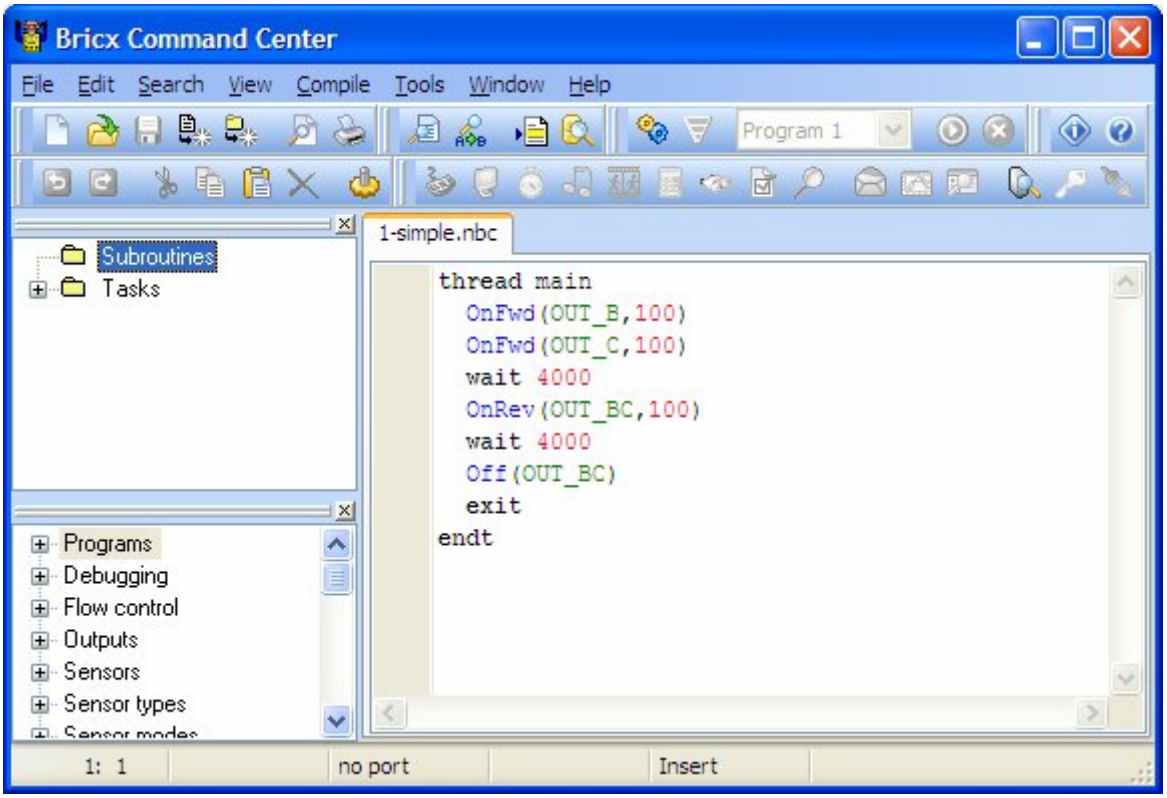
Building a robot

The robot we will use throughout this tutorial is the "Tribot", the instructions for which are included with your NXT set. If you are new to LEGO robotics, I recommend doing the tutorials that come with the LEGO software, to get acquainted with what your NXT can do. Your robot should look like this:

(Note that you may already have added sensors and a grabber – if so, you should remove them temporarily, as some of the examples may not work correctly with them attached.)

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 menus, 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:

```
thread main
  OnFwd(OUT_B,100)
  OnFwd(OUT_C,100)
  wait 4000
  OnRev(OUT_BC,100)
  wait 4000
  Off(OUT_BC)
  exit
endt
```

It might look a bit complicated at first, so let us analyze it. Programs in NBC consist of threads. Our program has just one thread, named `main`. Each program needs to have a thread called `main` which is the one that will be executed by the robot. You will learn more about threads in Chapter V. A thread consists of a number of commands, also called statements. Each statement takes a single line, so a task looks in general as follows:

```
thread main
  statement1
  statement2
endt
```

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

```
OnFwd(OUT_B,100)
```

This statement tells the robot to start output B, that is, the motor connected to the output labeled B on the NXT, to move forwards. The 100 specifies the percentage of maximum speed, so it will move with maximum speed.

```
OnFwd(OUT_C,100)
```
Same statement but now we start motor C. After these two statements, both motors are running, and the robot moves forwards.

```
wait 4000
```
Now it is time to wait for a while. This statement tells us to wait for 4 seconds. The argument gives the number of milliseconds, or 1/1000 of a second. So you can very precisely tell the program how long to wait. So for 4 seconds, the program does nothing and the robot continues to move forwards.

```
OnRev(OUT_BC,100)
```
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_BC` as argument. We could also have combined the first two statements this way. We could also use `OnFwd(OUT_BC,-100)`.

```
wait 4000
```
Again we wait for 4 seconds.

```
Off(OUT_BC)
```
And finally we switch both motors off.

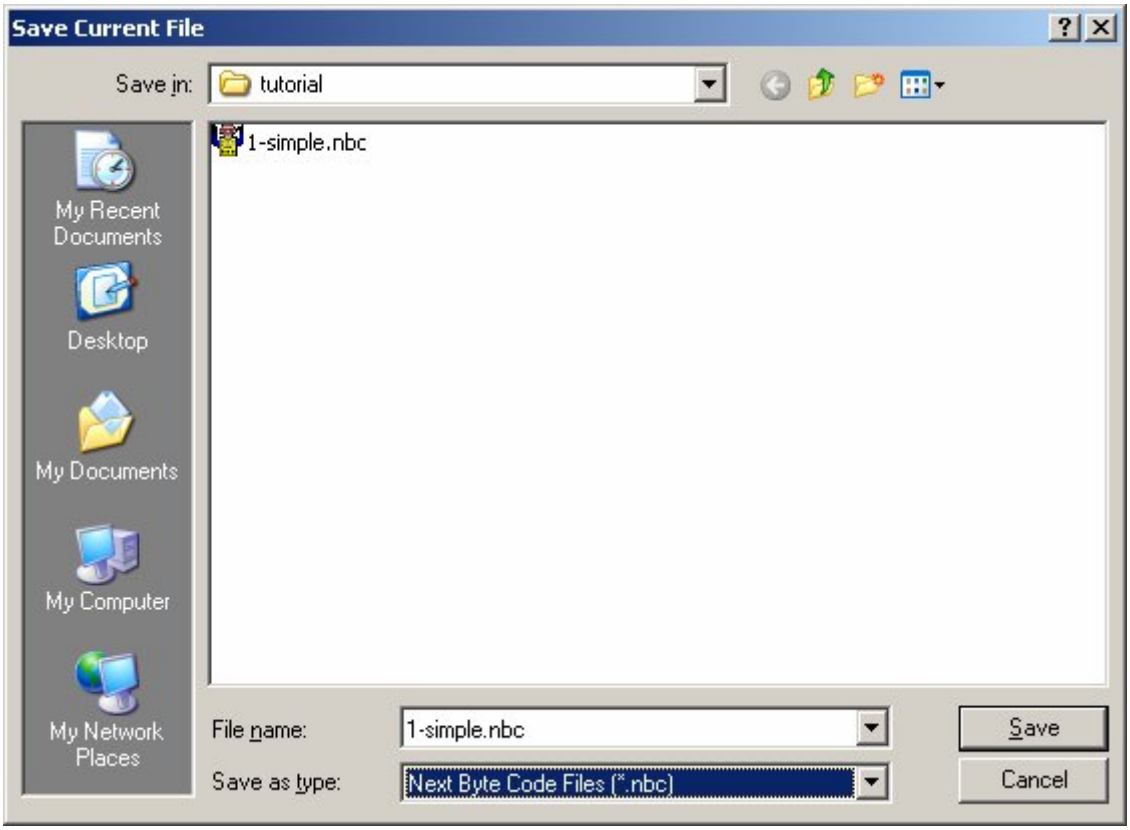
```
exit
```
This statement tells the NXT this thread has finished. While not required at the end of threads, it is recommended. Note also that it can appear elsewhere in the thread as well.

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 sent to the robot using either the USB cable or a Bluetooth device (called “downloading” the program). Before you can do that, you need to name the program, which you do by saving it to your hard drive. When you save it, make sure the file extension is “.nbc”, this tells Bricx Command Center that it is an NBC program.

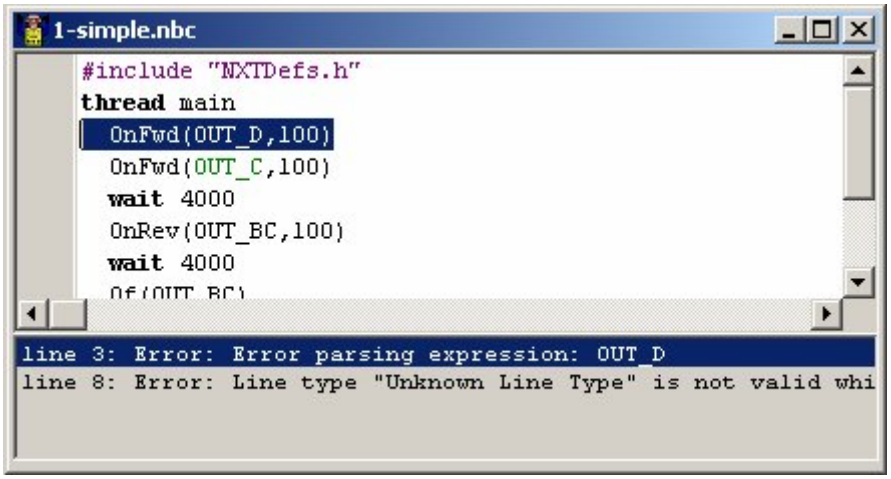


Once it has been saved, you can compile and download it simply by clicking the download button. Assuming you made no errors when typing in the program, it will correctly compile and be downloaded. (If there are errors in your program you will be notified; see below.)

Now you can run your program. To do this, go to “Software Files” on your NXT, and look for “1-simple”, then use the orange NXT button to run it. Or alternatively, you can run it from Bricx Command Center, by pressing the green run button on your window (see the figure above). Does the robot do what you expected? If not, the wires are probably connected incorrectly.

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, and you can spot the error even before you compile.

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, just use a different value in the OnFwd command. The power is a number between 0 and 100, and specifies the percentage of maximum power. Here is a new version of our program in which the robot moves slow:

```
thread main
  OnFwd(OUT_BC,25)
  wait 4000
  OnRev(OUT_BC,25)
  wait 4000
  Off(OUT_BC)
  exit
endt
```

In this example, you can see that the speed value, and the wait value are both repeated. If you needed to change these values, you would have to change them in several places, and you might miss one. So in NQC you can define constant values as shown in the following program.

```
#define SPEED        25
#define MOVE_TIME  4000

thread main
  OnFwd(OUT_BC, SPEED)
  wait MOVE_TIME
  OnRev(OUT_BC, SPEED)
  wait MOVE_TIME
  Off(OUT_BC)
  exit
endt
```

Now if you need to change the speed (for example), you only need to change it in one place.

Adding comments

To make your program even more readable, it is good to add some comments 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:

```
/*  Forward and reverse

    by Ross Crawford

This program makes the robot go forward and backward
*/

#define SPEED        25
#define MOVE_TIME  4000

thread main
  OnFwd(OUT_BC, SPEED)       // Drive forward
  wait MOVE_TIME             // Wait for 4 seconds
  OnRev(OUT_BC, SPEED)       // Drive backward
  wait MOVE_TIME             // Wait for 4 seconds
  Off(OUT_BC)                // Stop moving
  exit                       // Exit
endt
```

Summary

In this chapter you wrote your first program in NBC, 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 NBC and only mention features of Bricx Command Center when you really need them.

You also learned some important aspects of the language NBC. First of all, you learned that each program has one thread named `main` that is always executed by the robot. Also you learned the four most important motor commands: `OnFwd`, `OnRev` and `Off`, and about the `wait` and `exit` statement. Finally, you learned about how to use constants to make program-wide changes easier, and comments to explain what your code does.

II. 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 different ways

Assume we want to adapt the above program in such a way that the robot doesn't return as fast as it went forward. This can be achieved by making the speed value smaller for the return journey. But how can we do this? `SPEED` is a constant and constants cannot be changed. We need a variable instead. Variables can easily be defined in NBC. Here is a new program.

```
#define MOVE_TIME  4000
#define SPEED       100
#define DECREMENT    25

dseg segment
  Speed byte
dseg ends

thread main
  set Speed SPEED
  OnFwd(OUT_BC, Speed)
  wait MOVE_TIME
  sub Speed, Speed, DECREMENT
  OnRev(OUT_BC, Speed)
  wait MOVE_TIME
  Off(OUT_BC)
  exit
endt
```

We have introduced the variable `Speed` into the program. Every variable in your program must be declared in a data segment. You can have as many data segments as you want, and they can be pretty much anywhere in your program. All variables in NBC have global scope – they are accessible from any code in any thread.

Data segments start with the `segment` statement, and end with the `ends` statement. Each segment must be named, the name on the `segment` and `ends` statements must match.

So let's explain the new statements in this program.

`dseg segment`
This statement specifies the start of a data segment called "dseg". In our program, it is the only data segment and contains only a single variable.

`Speed byte`
This statement declares the variable "Speed" as type "byte". Variable names must start with a letter but can contain digits and the underscore sign. No other symbols are allowed. (The same applied to constants, thread names, etc.)

`dseg ends`
This statement specifies the end of the "dseg" data segment.

`set Speed, SPEED`
This statement assigns the value of SPEED (the constant defined earlier) to the variable Speed. Note that this can also be done in the declaration of the variable, by adding the initial value after the variable type, eg: `Speed byte SPEED`.

`sub Speed, Speed, DECREMENT`
This subtracts the value `DECREMENT` from the value currently stored in the variable `Speed`, and stores the result back to that same variable.

When you run this program, your robot should travel forward at full speed for the specified time, then reverse for the same time at a slower speed.

Besides subtracting values from a variable we can also multiply a variable with a number, subtract and divide. (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:

```
dseg segment
  aaa byte
  bbb byte
  ccc byte
dseg ends

thread main
  set aaa, 10          // aaa is now equal to 10
  mul bbb, 20, 5       // bbb is now equal to 100
  mov ccc, bbb         // ccc is now equal to 100
  div ccc, ccc, aaa    // ccc is now equal to 10
  add ccc, ccc, 5      // ccc is now equal to 15

  exit
endt
```

Now let's look at the new code more closely.

`set aaa, 10`
We've seen this one before, but I just wanted to point out that the second argument to `set` can only be a constant. This is different from the `mov` statement below.

`mul bbb, 20, 5`
This statement multiplies 20 by 5, and assigns the result to the variable `bbb`. The 2nd and 3rd arguments can be either constants or other variables.

`mov ccc, bbb`
This statement assigns the value of the variable `bbb` to the variable `ccc`. Unlike set, the 2nd argument of `mov` can be either a constant or a variable.

`div ccc, ccc, aaa`
This statement divides `ccc` by `aaa` and assigns the result back to `ccc`. Note that it returns integers only – any remainder is truncated.

`add ccc, ccc, 5`
This statement add 5 to `ccc`.

Displaying results on the screen

As you can see, if perform these commands, the final value of ccc should be 15. But how do we check it? The easiest way is to display the result on the NXT screen. This is fairly simple – there is a system call available to do it. The following program demonstrates it:

```
dseg segment
  aaa byte
  bbb byte
  ccc byte
dseg ends

thread main
  set aaa, 10          // aaa is now equal to 10
  mul bbb, 20, 5       // bbb is now equal to 100
  mov ccc, bbb         // ccc is now equal to 100
  div ccc, ccc, aaa    // ccc is now equal to 10
  add ccc, ccc, 5      // ccc is now equal to 15

  NumOut(10, LCD_LINE7, ccc)
  wait 2000                  // wait so you get to see it!

  exit
endt
```

This is fairly self-explanatory, but here is an explanation of the important parts.

`NumOut(10, LCD_LINE7, ccc)`
This statement converts the value `ccc` to a string, then calls the system function `DrawText` to display the string on the screen. The value of `ccc` should be displayed near the lower left of the screen.

`wait 2000`
We've seen this before too – it is here simply to pause and allow you to see the result. If it wasn't here, the program would exit immediately, and the NXT menu would overwrite the value you just displayed too fast for you to see it!

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 NBC you can create random numbers. The following program uses this to pick a random speed when it reverses.

```
#define MOVE_TIME  4000
#define SPEED       100
#define DECREMENT    25

dseg segment
  Speed byte
  wRandom byte
dseg ends

thread main
  set Speed SPEED
  OnFwd(OUT_BC, Speed)
  wait MOVE_TIME
  Random(wRandom,30)
  add wRandom, wRandom, 10
  sub Speed, Speed, wRandom
  OnRev(OUT_BC, Speed)
  wait MOVE_TIME
  Off(OUT_BC)
  exit
endt
```

The program is basically the same as the one above, but instead of just subtracting 25 from Speed, it uses a random number between 10 and 40. Let's look at how it does this:

`Random(wRandom,30)`
This statement generates a random integer between 0 and 29 (inclusive), and returns the result in `wRandom`. Note that the result will always be less then the value of the 2nd parameter.

`add wRandom, wRandom, 10`
This statement adds 10 to the result, so it will now be between 10 and 39.

`sub Speed, Speed, wRandom`
This statement subtracts the result from the current value of `Speed`, ready to change the random speed in the following `OnRev` statement.

Summary

In this chapter you learned about the use of variables. Variables are very useful, but they are restricted to only integer values. But for many robot tasks this is good enough.

You also learned how to display a value on the NXT screen. This technique can be very useful for debugging more complex programs.

Finally, you learned how to create random numbers, such that you can give the robot unpredictable behavior.

III. Flow Control

In the previous chapters we saw many ways to manipulate variables. But all the statements we've looked at so far just get executed in order, which is great for some things, but often we need to make decisions about what to do based on different conditions. These require a way to compare variables, and do different things based on the result of the comparison.

The cmp and tst statements

Sometimes you want to just record the result of a comparison so you can use that result later in the program. This can easily be accomplished using the `cmp` and `tst` statements. Let's look at how we use these in a program.

```
dseg segment
  aaa byte
  bbb byte
  ccc byte
  wRandom byte
dseg ends

thread main
  set aaa, 10
  Random(wRandom,20)
  mov bbb, wRandom
  cmp GT, ccc, bbb, aaa

  NumOut(10, 8, bbb)
  NumOut(50, 8, ccc)

  wait 2000

  exit
endt
```

This program should be fairly familiar – it is based on the one in the previous chapter. However, there is one new statement: let's have a look at it.

`cmp GT, ccc, bbb, aaa`
This statement compares `bbb` with `aaa` using the comparison `GT`, which means "greater than". Then, the result is stored in `ccc`. So, if `bbb` is greater than `aaa`, the value 1 (which is how the NXT firmware represents "true") is stored in `ccc`, otherwise the `ccc` is assigned the value 0.

So this does exactly what we wanted – it stores the result of the comparison in a variable for later. The lines following the comparison write the values of bbb and ccc to the display so you can see how the comparison works. Go ahead, run the program a few times, and verify that the value of ccc is set correctly depending on the random value of bbb.

So what about the `tst` statement? Well it's really just the poorer brother of `cmp`, it works the same except it only has 3 parameters, and assumes a value of zero for the fourth parameter. So the following statements would do the same thing:

```
  cmp GT, ccc, bbb, 0
  tst GT, ccc, bbb
```

There are other types of comparison that can be performed with `cmp` and `tst`, too. Here is the list:

EQ	equal to
LT	smaller than
LTEQ	smaller than or equal to
GT	larger than
GTEQ	larger than or equal to
NEQ	not equal to

The brcmp and brtst statements

So, we can store the result of a comparison for later using `cmp` or `tst`. And we can display it on the screen. But how can we actually use it to do something? Generally, a robot will want to do something different depending on the result of the test, and this can be accomplished with `brcmp` and `brtst`.

In NBC, all program branch statements require a label. The label is just a string of characters followed by a colon. A label can be on a line by itself or at the start of a line containing an NBC statement. Let's have a look at an example:

```
dseg segment
  aaa byte
  bbb byte
  wRandom byte
dseg ends

thread main
  set aaa, 10
  Random(wRandom,20)
  mov bbb, wRandom

  NumOut(10, 8, bbb)

  brcmp GT, Bigger, bbb, aaa
  TextOut(50, 8, 'small')
  brcmp LTEQ, Delay, bbb, aaa
Bigger:
  TextOut(50, 8, 'BIG')
Delay:
  wait 2000

  exit
endt
```

Again, this is fairly similar to the last program, but instead of using `cmp` to store the comparison result, we use `brcmp` to execute some "conditional" code. Let's analyse the 2 new statements:

`brcmp GT, Bigger, bbb, aaa`
This looks very similar to the cmp statement in the last program doesn't it? In fact it is, and the exact same comparison is performed, but instead of storing the result, it is used to determine whether or not to skip program execution to the label "Bigger", which is defined below. If the comparison is true, the program skips, otherwise it continues with the next statement, which displays the string "small".

`Bigger:`
This is a label, and is the target of the `brcmp` statement. If the comparison is true, program execution will skip to the first statement following this label, which displays the string "BIG".

So the program will display the value of `bbb`, and either "BIG" or "small" depending whether or not it's bigger than `aaa`.

The brtst statement is, again, similar to brcmp, except that the 4th parameter is assumed to be zero.

The jmp statement

Sometimes you need to do an unconditional branch, that is, skip some code no matter what. This can be accomplished with the `jmp` statement. It is very simple; you just specify the label you want to skip to.

One common use of unconditional branches is in an "if-then-else" type of situation. You use brcmp or brtst as the "if", jumping to a label to execute the "then" code. If the comparison is false, you execute the "else" code, immediately below the conditional branch. But at the end of that code, you want to skip over the "then" code, because you don't want to execute that if the test is false. So you must add a label after the "then" code, and skip to it unconditionally at the end of the "else" code. Let's look at an example:

```
#define MOVE_TIME  4000
#define SPEED       100
#define DECREMENT    25

dseg segment
  Speed byte
  wRandom byte
dseg ends

thread main
  set Speed SPEED
  OnFwd(OUT_BC, Speed)
  wait MOVE_TIME
  sub Speed, Speed, DECREMENT
  Random(wRandom,2)
  brtst EQ, Then, wRandom
Else:
  OnRev(OUT_BC, Speed)
  jmp EndIf
Then:
  OnFwd(OUT_BC, Speed)
EndIf:
  wait MOVE_TIME
  Off(OUT_BC)
  exit
endt
```

Here is an example “if-then-else”. In this example, our robot should go forward at full speed for 4 seconds, then go either forwards or reverse at slower speed for another 4 seconds, depending on a random number. Let’s analyze it:

`brtst EQ, Then, wRandom`
This is the conditional branch, or “if” statement. If `wRandom` is equal to zero, program execution will skip to the “Then” label.

`Else:`
This is a label identifying the “Else” clause, that is the code which is executed if the comparison is false. Note that this label is in fact unnecessary, as no branch statement uses it as a target. It was included to highlight the similarity to an if-then-else construct, and also to demonstrate that it is OK to put labels in your code which are not the target of branch statements. So if the test is false, the OnRev statement will be executed, and the robot will back up.

`jmp EndIf`
This unconditional branch causes program execution to skip to “Endif”, thus avoiding execution of the “Then” clause.

`Then:`
The “Then” label is the target of the `brtst` statement above, so if the test is true, program execution will skip to this point, thus missing the “Else” clause, and the robot will continue forwards, due to the `OnFwd` statement.

`EndIf:`
The “Endif” label is the target of the unconditional branch at the end of the “Else” clause.

If you run this program a few times, your robot should go forward about half the time and backward about half the time.

Loops – repeating code

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, save it, download it to your robot and let it run. It should drive a bit and then make a 90-degree right turn.

```
#define MOVE_TIME   800
#define TURN_TIME   200

thread main
  OnFwd(OUT_BC,100)
  wait MOVE_TIME
  OnRev(OUT_C,100)
  wait TURN_TIME
  Off(OUT_BC)
endt
```

You might have to try some slightly different numbers than 200 for the value of TURN_TIME to make a precise 90-degree turn. This depends on the type of surface on which the robot runs

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 a loop. Loops in NBC make use of the same branch statements and we used earlier.

```
#define MOVE_TIME   800
#define TURN_TIME   200

dseg segment
  SquareCount byte 4
dseg ends

thread main

SquareLoop:
  OnFwd(OUT_BC,100)
  wait MOVE_TIME
  OnRev(OUT_C,100)
  wait TURN_TIME
  sub SquareCount, SquareCount, 1
  brtst GT, SquareLoop, SquareCount

  Off(OUT_BC)
endt
```

This program declares a variable `SquareCount`, and initializes its value to 4. Then it executes the same code we introduced in the previous example, to move forward, then turn 90 degrees. After that, it subtracts 1 from `SquareCount`, and then compares the new value with zero. If `SquareCount` is greater than zero, it branches to the label `SquareLoop`, thus going back and re-executing the movement commands again. After 4 times through the “loop”, `SquareCount` reaches zero, and the motors are switched off before exiting the program.

So the robot should drive around the sides of a square (approximately), and return to where it started.

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

```
#define MOVE_TIME   800
#define TURN_TIME   200

dseg segment
  SquareCount byte 4
  RepeatCount byte 10
dseg ends

thread main

RepeatLoop:
  set SquareCount, 4

SquareLoop:
    OnFwd(OUT_BC,100)
    wait MOVE_TIME
    OnRev(OUT_C,100)
    wait TURN_TIME
    sub SquareCount, SquareCount, 1
    brtst GT, SquareLoop, SquareCount

  sub RepeatCount, RepeatCount, 1
  brtst GT, RepeatLoop, RepeatCount

  Off(OUT_BC)
endt
```

There is now one loop inside the other. We call these “nested” loops. You can nest loops as much as you like. Notice that each loop is indented – this is not necessary, but helps to make the program easier to read. Note also that although `SquareCount` is initialized in the variable declaration, it needs to be reset to 4 before each time through the `SquareLoop`. This is necessary because the previous execution of that loop leaves the value at zero.

Summary

In this chapter you learned about how to compare values in your program and store the result, using the `tst` and `cmp` statements. You learned how to conditionally skip code using labels and the `brtst` and `brcmp` statements, and unconditionally skip code using the `jmp` statement. Finally, you learned how to use the `brtst`, `brcmp` and `jmp` statements to write if-then-else clauses and loops.

IV. 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 do this, you will need to follow the instructions in the software that came with your NXT. You need to build the "bumper" for you tribot, according to those instructions, and your robot should now look like this:

Connect the sensor to input 1 on the NXT.

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:

```
dseg segment
  Switch sword 0
dseg ends

thread main
  SetSensorTouch(IN_1)
  OnFwd(OUT_BC,100)

CheckSensor:
  ReadSensor(IN_1,Switch)
  brtst EQ, CheckSensor, Switch

  Off(OUT_BC)
endt
```

Let's look at the important lines:

`SetSensorTouch(IN_1)`
This tells the NXT that the sensor connected to the input `IN_1` will be a touch sensor.

`ReadSensor(IN_1,Switch)`
This reads the current value of the sensor, translates it to an appropriate value depending on the sensor type selected, and returns it in the 2nd parameter. So in our program, Switch will contain either 1 or 0 (true/false) after this call, depending whether the touch sensor is pressed or not.

`brtst EQ, CheckSensor, Switch`
This causes the program to loop until the value of Switch is not equal to zero. So it will just keep checking the touch sensor and looping until something activates it. As soon as the touch sensor is pressed (by the robot running into something), the loop is exited, and the robot stops.

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:

```
dseg segment
  Switch sword 0
dseg ends

thread main
  SetSensorTouch(IN_1
  OnFwd(OUT_BC,100)

CheckSensor:
  ReadSensor(IN_1,Switch)
  brtst EQ, CheckSensor, Switch
  OnRev(OUT_BC,100)
  wait 300
  OnFwd(OUT_B,100)
  wait 300
  OnFwd(OUT_BC,100)
  jmp CheckSensor

endt
```

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 a touch sensor, 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. To do that, follow the instructions in the program supplied with your NXT. Make sure it is connected to input 3. Your robot should now look like this:

We also need the race track that comes with your NXT 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 60

dseg segment
  Level sword 0
dseg ends

thread main
  SetSensorLight(IN_3)
  OnFwd(OUT_BC,100)

CheckSensor:
  ReadSensor(IN_3,Level)
  brcmp LT, CheckSensor, Level, THRESHOLD
  OnRev(OUT_B,50)

FindLine:
  ReadSensor(IN_3,Level)
  brcmp GTEQ, FindLine, Level, THRESHOLD

  OnFwd(OUT_BC,100)
  jmp CheckSensor

endt
```

The program first indicates that sensor 3 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 60 (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 use a second loop to 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 100` command after the `OnRev` 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.

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 make a robot that stays within an area indicated by a thick black border line on the floor.

V. Making music

The NXT 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 NXT tell you that something is happening. But it can also be funny to have the robot make music while it runs around.

Playing tones

For basic music, NBC has the command `PlayTone()`. It has two arguments. The first is the frequency, and the second the duration (in ticks of 1/1000th of a second, like in the wait command). There are 5 octaves of tones defined in the NBC API, from C3 to B7, but not all those values produce a useful note.

```
thread main
  PlayTone(TONE_C5,500)
  wait 500
  PlayTone(TONE_D5,500)
  wait 500
  PlayTone(TONE_E5,500)
  wait 500
  PlayTone(TONE_D5,500)
  wait 500
  PlayTone(TONE_C5,1000)
  wait 1000
endt
```

You might wonder why there are wait commands between each call to `PlayTone`. The reason is that the command that plays the sound does not wait for it to finish. It immediately executes the next command. The NXT 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.

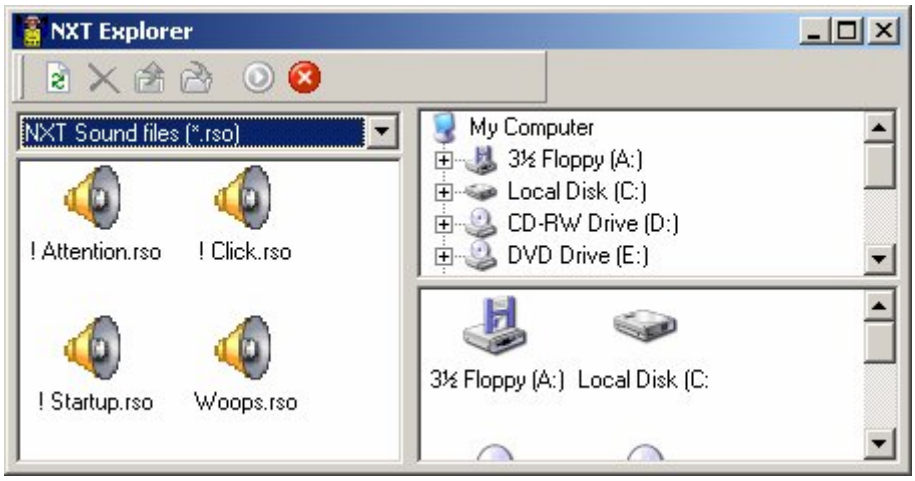
Note that the arguments to `PlayTone()`can be either constants or variables. So you could write a loop in your program and increment or decrement either the frequency or the duration using the math opcodes discussed in Chapter II.

```
dseg segment
  freq word TONE_C3
  loopCount byte 4
dseg ends

#define DURATION 500

thread main
  DoLoop:
    PlayTone(freq, DURATION)
    wait DURATION
    mul freq, freq, 2
    sub loopCount, loopCount, 1
  brtst GT, DoLoop, loopCount
  // the loop is finished, so play our last tone
  PlayTone(TONE_C5, DURATION*2)
  wait DURATION*2
  exit
endt
```

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

Playing files

As well as playing tones, NBC has the command `PlayFile()`, which allows you to play sound samples and NXT melody files. It has one argument, being the name of the file to play. You can view what sound files and melody files are available on the NXT using the NXT Explorer tool included with Bricx Command Center. Choosing NXT Explorer from the Tools menu opens a window like this:



By choosing "NXT Sound files" from the combo list, you can display only the sound files available on the NXT and on your PC. The 4 files displayed are already on your NXT when the standard firmware is installed. We can write a very simple program to play the Woops.rso file like this:

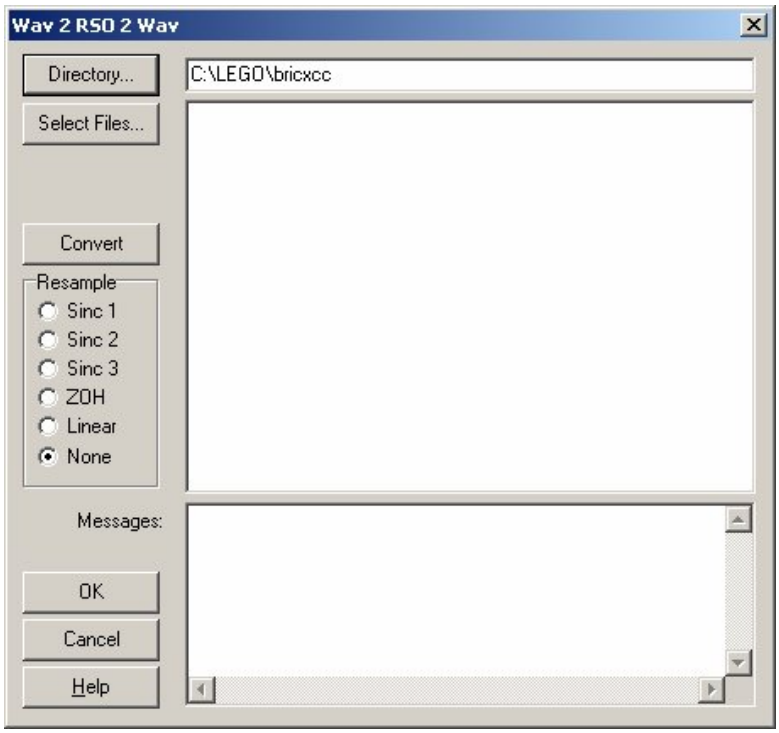
```
dseg segment
  MyFile byte[] 'Woops.rso'
dseg ends

thread main
  PlayFile(MyFile)
  wait 1000
endt
```

Again, you can see we need a `wait` command after the `PlayFile()` command, for the same reason as before – `PlayFile()` does not wait for the sample to finish before returning control to your program. Also, note that you must include the complete filename, but it is not case sensitive. If the file cannot be found, the command will be silently ignored. You can also simply pass the filename directly to the `PlayFile()` command without creating a variable first.

To play a melody file you would use the same sort of code, except that melody files normally have a ".rmd" file extension rather than ".rso". The NXT does not come with any melody files already on the brick, but you can easily create them either by using the Brick Piano in BricxCC or by using the MIDI conversion tool in BricxCC.

Creating your own sound files

The built in sounds are fine for a lot of things, but what if you want to create your own custom sounds? BricxCC also includes a tool to convert Windows™ WAV files to the RSO format used by NXT. Choosing "Sound Conversion…" from the Tools menu opens a window similar to this:



You can use the Select Files button to select any WAV file on your PC, then use the Directory button to choose where to save the converted file. Then click "Convert" to convert the file – it will be saved in the directory you chose, with the same name as the original WAV file, but with an extension of RSO instead of WAV. The Resample options can reduce the size of the resulting file, but may also reduce the quality of the resulting sound. But if memory is tight, you should experiment with these options to find the best compromise. The result of the conversion will be displayed in the Messages window at the bottom of the dialog.

After you have created your sound file, you can transfer it to the NXT using the NXT Explorer tool, and your program will then be able to play it exactly as we did earlier with the built-in file. Downloading a sound file from your computer to the NXT using the NXT Explorer tool is accomplished by simply dragging the file from the PC window and dropping it on the NXT window.

Another source for sound files is the standard NXT software. It comes with many different sounds. Using NXT Explorer in BricxCC you can browse on your PC to the Program Files\LEGO Software\LEGO MINDSTORMS NXT\engine\Sounds directory and drag sound files to the NXT. Sound files can use a lot of memory on your NXT so be careful how many you use. Melody files are usually much smaller than sound files so they make a nice alternative to use in your program.

Summary

In this chapter you learned how to let the NXT make basic music and play sound or melody files. Finally, you learned how to create your own sound files or melody files and download them to your NXT.

VI. Threads and subroutines

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

Threads

As explained earlier, an NBC program consists of at least 1 thread. Each thread has a name. One thread should have the name `main`, and this thread will be executed first. The other threads will only be executed when a running thread tells them to be executed. There are 2 ways to start another thread – using the precedes and follows commands, or the exitto command.

Let me demonstrate the use of threads. Say we want to take the square-moving robot from chapter III, and have it play music while it moves. We need to do 2 things simultaneously, and that is exactly what threads allow you to do. So what we can do is have 1 thread controlling the motors, and another thread playing the music, then get them to execute simultaneously. Here is how we do that:

```
#define MOVE_TIME   800
#define TURN_TIME   200

thread main
  precedes move_square, play_music
endt

thread move_square
SquareLoop:
  OnFwd(OUT_BC,100)
  wait MOVE_TIME
  OnRev(OUT_C,100)
  wait TURN_TIME
  jmp SquareLoop
endt

thread play_music
MusicLoop:
  PlayTone(TONE_C5,500)
  wait 500
  PlayTone(TONE_D5,500)
  wait 500
  PlayTone(TONE_E5,500)
  wait 500
  PlayTone(TONE_D5,500)
  wait 500
  jmp MusicLoop
endt
```

The main thread just starts both other tasks, then exits immediately – its job is done. Thread `move_square` moves the robot forever in squares. Thread `play_music` just plays the same sequence of 4 tones over and over.

It is very important to remember that tasks that you start are running at the same moment. This can lead to unexpected results. Chapter IX 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 thread. Let us look at an example.

```
#define TURN_TIME   340

thread main
  OnFwd(OUT_BC,100)
  wait 1000
  call turn_around
  wait 2000
  call turn_around
  wait 1000
  call turn_around
  Off(OUT_BC)
endt

subroutine turn_around
  OnRev(OUT_C,50)
  wait TURN_TIME
  OnFwd(OUT_BC,100)
  return
ends
```

In this program we have defined a subroutine that makes the robot rotate around its center. The main thread calls the subroutine three times. The subroutine is declared between the `subroutine` and `ends` keywords, and must be named. It can have `return` commands anywhere within it, but should always have one at the end. It is executed using the `call` command, and after it has completed, control automatically returns to the next command after the `call` that invoked it.

Some warnings are in place here. Subroutines are a bit weird. First, subroutines cannot have parameters, like other commands. Subroutines can be called from different threads 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. So, unless you know precisely what you are doing, *don't call a subroutine from different threads!*

NBC (or actually the NXT) allows for at most 255 threads *and* subroutines combined.

Defining macros

There is yet another way to give small pieces of code a name. You can define macros in NBC. 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_TIME   340
#define turn_around \
  OnRev(OUT_C,50)   \
  wait TURN_TIME    \
  OnFwd(OUT_BC,100)

thread main
  OnFwd(OUT_BC,100)
  wait 1000
  turn_around
  wait 2000
  turn_around
  wait 1000
  turn_around
  Off(OUT_BC)
endt
```

After the `#define` statement the word `turn_around` stands for the text after it. Now wherever you type `turn_around`, this is replaced by this text. Note that multiple commands can be included, but each except the last must be followed by the backslash (\) character.

Define statements are actually a lot more powerful. They can 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 macros; one to move

forwards, one to move backwards, one to turn left and one to turn right. Each has two arguments: the power and the time.

```
#define turn_right(pwr,time) \
  OnRev(OUT_C,pwr)   \
  wait time          \
  OnFwd(OUT_B,pwr)

#define turn_left(pwr,time) \
  OnRev(OUT_B,pwr)   \
  wait time          \
  OnFwd(OUT_C,pwr)

#define forwards(pwr,time) \
  OnFwd(OUT_BC,pwr)   \
  wait time

#define backwards(pwr,time) \
  OnRev(OUT_BC,pwr)   \
  wait time

thread main
  forwards(50,2000)
  turn_left(100,850)
  forwards(100,1000)
  backwards(100,2000)
  forwards(100,1000)
  turn_right(100,850)
  forwards(50,2000)
  Off(OUT_BC)
endt
```

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 threads, subroutines and macros. They have different uses. Threads normally run at the same time and take care of different things that have to be done at the same time. Subroutines are useful when larger pieces of code must be used at different places in the same thread. 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. 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 NBC it is also possible to stop the motors in a more gentle way, not using the brake. For this you use the `Coast()` 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.)

```
thread main
  OnFwd(OUT_BC,100)
  wait 200
  Off(OUT_BC)
  wait 100
  OnFwd(OUT_BC,100)
  wait 200
  Coast(OUT_BC)
endt
```

Synchronising motors

One big problem with robots that have a separate motor for each wheel is that driving in a perfectly straight line can be difficult. If the motors run at just slightly different speeds, your robot will tend to drive in a large circle when you want it to go straight. The NXT is able to overcome this problem by using motors with rotation sensors built in, allowing you to synchronise them. You can use the `OnFwdReg()` and `OnRevReg()` commands to easily do this:

```
thread main
  OnFwdReg(OUT_BC, 100, OUT_REGMODE_SYNC)
  wait 2000
  Off(OUT_BC)
  wait 100
  OnRevReg(OUT_BC, 100, OUT_REGMODE_SYNC)
  wait 2000
  Coast(OUT_BC)
endt
```

The NXT will ensure that the motors remain synchronized; if one of them encounters something that causes it to slow down, the other will slow down to compensate.

Regulating the motor speed

Another problem with robot motors is that setting a fixed power doesn't mean the speed is fixed – the robot will slow down if the load on it is increased. The special NXT motors provide a solution to this as well, using the same functions as in the previous section:

```
thread main
  OnFwdReg(OUT_BC, 100, OUT_REGMODE_SPEED)
  wait 2000
  Off(OUT_BC)
  wait 100
  OnRevReg(OUT_BC, 100, OUT_REGMODE_SPEED)
  wait 2000
  Coast(OUT_BC)
endt
```

The NXT will automatically adjust the power to the motors to attempt to keep the speed constant. Note that there are limitations to this; if the load is increased too much, the NXT will not be able to supply enough power to the motors and they will slow down.

Note also that the OUT_REGMODE parameters can be combined, eg: `OnFwdReg(OUT_BC, 100, OUT_REGMODE_SYNC+OUT_REGMODE_SPEED)` and the NXT will keep the motors synchronized as well as attempting to keep their speed constant.

Rotating a specific angle

Another function often required in robots is the ability to rotate a motor by a specific angle. The special NXT motors also make this fairly simple:

```
thread main
  RotateMotor(OUT_A, 100, 90)
endt
```

This code will rotate motor A by 90 degrees, at 100% power. You can specify an angle greater than 360, and it will turn more than a full turn, if you specify a negative angle it will turn in reverse. There are a couple of things to remember about this function though:

1. Unlike most motor control commands so far, you can only specify a single motor in this command – you cannot rotate 2 motors with a single `RotateMotor()` command;
2. Depending on the power parameter, the motor may initially overshoot the desired angle, and will toggle direction until it returns to the correct position.

This function uses a PID (proportional, integral, differential) algorithm to control the motor angle, and for those who know what they're doing, the PID parameters can be adjusted using the advanced motor control functions described below.

More advanced motor control

The above functions are high-level macros to make using common features easy. But NBC has low-level motor control functions, and if your robot has advanced requirements, you may need to use these functions. I will provide only a brief description of these functions here – for more detailed information, please refer to the NBC manual.

The `setout` command sets one or more output fields of a motor on one or more ports to the value specified by the coupled input arguments.

```
setout port/portlist, fieldid1, value1, …, fieldidN, valueN
```

```
setout OUT_A, OutputMode, OUT_MODE_MOTORON, RunState, OUT_RUNSTATE_RUNNING,
Power, 75
```

The `getout` command reads a value from an output field of a sensor on a port and writes the value to its dest output argument.

```
getout dest, port, fieldID
```

```
getout myVar, OUT_A, OutputMode
```

Output Field Identifiers

UpdateFlags, OutputMode, Power, ActualSpeed, TachoCount, TachoLimit, RunState, TurnRatio, RegMode, Overload, RegPValue, RegIValue, RegDValue, BlockTachoCount, RotationCount

Summary

In this chapter you learned about the extra high-level motor commands that are available: `Coast()` that stops the motor gently, `OnFwdReg()` and `OnRevReg()` that allow you to regulate your motor speed and synchronization, and `RotateMotor()` that allows you to control the angle of the motor shaft. You also learned about the low-level commands for controlling and interrogating motor parameters.

VIII. More about sensors

In Chapter IV 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 sound sensor and ultrasonic sensor included with your NXT.

Sensor type

The `SetSensorTouch()`and `SetSensorLight()` command that we saw before does actually three things: it sets the type of the sensor, and it sets the mode in which the sensor operates, then resets the sensor. By setting the mode and type of the 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()`.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. Here is a list of valid sensor types:

- IN_TYPE_NO_SENSOR – No sensor attached.
- IN_TYPE_SWITCH – Standard LEGO® NXT touch sensor attached.
- IN_TYPE_TEMPERATURE – Standard LEGO® NXT temperature sensor attached.
- IN_TYPE_REFLECTION – Standard LEGO® NXT touch sensor attached.
- IN_TYPE_ANGLE – Standard LEGO® NXT rotation sensor attached.
- IN_TYPE_LIGHT_ACTIVE – Standard LEGO® NXT light sensor attached – lamp will be activated.
- IN_TYPE_LIGHT_INACTIVE – Standard LEGO® NXT light sensor attached – lamp will not be activated.
- IN_TYPE_SOUND_DB – Standard LEGO® NXT sound sensor attached – will measure dB.
- IN_TYPE_SOUND_DBA – Standard LEGO® NXT sound sensor attached – will measure dBa.
- IN_TYPE_CUSTOM – Currently unused.
- IN_TYPE_LOWSPEED – I2C sensor attached – unpowered.
- IN_TYPE_LOWSPEED_9V – I2C sensor attached – powered (9V).
- IN_TYPE_HISPEED – Currently unused.

Sensor mode

The mode of the sensor is set with the command `SetSensorMode()`. There are ten different modes. The most important one is `IN_MODE_RAW`. In this mode, the value you get when using `ReadSensor()`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 180. When the sensor is a light sensor, the value ranges from about 420 (very light) to 270 (very dark). This can give a much more precise value than using the `SetSensorLight()` command. Here is a list of sensor scaling modes:

- IN_MODE_RAW – No scaling of the raw value.
- IN_MODE_BOOLEAN – Value scaled to 1 (TRUE) or 0 (FALSE). Readings are FALSE if raw value exceeds 55% of total range; readings are TRUE if raw value is less than 45% of total range.
- IN_MODE_TRANSITIONCNT – Value returned as number of transitions between TRUE and FALSE.
- IN_MODE_PERIODCOUNTER – Value returned as number of transitions from FALSE to TRUE, then back to FALSE.
- IN_MODE_PCTFULLSCALE – Value scaled as percentage of full scale reading for configured sensor type.
- IN_MODE_CELSIUS – Scale TEMPERATURE reading to degrees Celsius.
- IN_MODE_FAHRENHEIT – Scale TEMPERATURE reading to degrees Fahrenheit.
- IN_MODE_ANGLESTEP – Value returned as count of ticks on RCX-style rotation sensor.
- IN_MODE_SLOPEMASK
- IN_MODE_MODEMASK

There are two other interesting modes: `IN_MODE_TRANSITIONCNT` and `IN_MODE_PERIODCOUNTER`. 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 `IN_MODE_PERIODCOUNTER`, 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 `IN_MODE_TRANSITIONCNT`, 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.

Sound sensor

The NXT set also includes a sound sensor. This allows you to program your robot to do things based on how loud the ambient sound is. Note that it only provides a total volume sample; it doesn't provide any sound processing abilities.

Let us look at an example. The following program uses the sound sensor to steer the robot. If you clap quickly twice the robot moves forwards. It you clap once it stops moving.

```
dseg segment
  Count sword 0
dseg ends

thread main
  SetSensorType(IN_2, IN_TYPE_SOUND_DB)
  SetSensorMode(IN_2, IN_MODE_PERIODCOUNTER)
  ResetSensor(IN_2)

Forever:
  ClearSensor(IN_2)
CheckCount:
  ReadSensor(IN_2, Count)
  brtst EQ, CheckCount, Count
  wait 500
  ReadSensor(IN_2, Count)
  brcmp EQ, StopIt, Count, 1
  OnFwd(OUT_BC,100)
  jmp Forever
StopIt:
  Off(OUT_BC)
  jmp Forever
endt
```

This code also demonstrates the use of a non-standard sensor mode. Normally we would use the `SetSensorSound()` to initialize the sound sensor, but in this case, we only want to measure volume transitions, so we use the period counter mode. 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 affects the mode. Note also the use of the `ResetSensor()` command. This is required after you set the type or the mode.

Motor as a rotation sensor

As well as using the feedback functions of the motor for speed control and synchronization, it can be used as a stand-alone rotation sensor by connecting it to one of the sensor inputs. The downside is that it requires quite a large force to turn it, as you also have to turn the motor and associated gearing. But for certain applications, this may be useful.

To be completed.

Ultrasonic sensor

The last sensor included with the NXT set is an ultrasonic sensor. This is a sensor that sends out ultrasonic pulses, and measures how long it takes for their reflection to return, thus giving an approximate measure of distance to the nearest object.

This sensor is different from the ones we've used so far, in that it uses the digital I2C sensor interface. This means it isn't possible to communicate with it using the standard `ReadSensor()` command; a separate `ReadSensorUS()` command is provided instead. Apart from that, using it is pretty much the same as any other sensor. Here is an example that lets the robot run forwards until it gets near to an object and then makes a 90 degree turn to the right.

```
dseg segment
  Distance sword 0
dseg ends

thread main
  SetSensorUltrasonic(IN_4)
  OnFwd(OUT_BC,100)

Forever:
  ReadSensorUS(IN_4, Distance)
  brcmp GT, Forever, Distance, 30
  Off(OUT_B)
  wait 200
  OnFwd(OUT_BC,100)
  jmp Forever
endt
```

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.

More advanced sensor control

The above functions are high-level macros to make using common features easy. But NBC has low-level sensor control functions, and if your robot has advanced requirements, you may need to use these functions. I will provide only a brief description of these functions here – for more detailed information, please refer to the NBC manual.

The `setin` command sets an input field of a sensor on a port to the value specified in its first input argument.

```
setin value, port, fieldID
```

```
      setin IN_TYPE_SWITCH, IN_1, Type
      setin IN_MODE_BOOLEAN, IN_1, InputMode
```

The `getin` command reads a value from an input field of a sensor on a port and writes the value to its dest output argument.

```
getin dest, port, fieldID
```

```
      getout myVar, IN_1, ScaledValue
```

Input Field Identifiers

Type, InputMode, RawValue, NormalizedValue, ScaledValue, InvalidData

Putting it all together

Here is a project from the LEGO® NXT software that uses everything you've learned in the previous 2 chapters about motors and sensors. It assumes you have completed building the Tribot; it should now look something like this:

This program will wait until a ball is placed in front of the Tribot, when it will move forward until it touches the ball then stop. When you clap your hands, it will close the grabber, turn around, and return to the starting point, where it will open the grabber and drop the ball.

```
dseg segment
  Switch sword 0
  Level sword 0
  Volume sword 0
  Distance sword 0
dseg ends

thread main
  SetSensorTouch(IN_1)
  SetSensorSound(IN_2)
  SetSensorLight(IN_3)
  SetSensorUltrasonic(IN_4)

CheckObject:
  ReadSensorUS(IN_4, Distance)
  brcmp GT, CheckObject, Distance, 20

  OnFwdReg(OUT_BC,75,OUT_REGMODE_SYNC)

CheckTouch:
  ReadSensor(IN_1, Switch)
  brtst EQ, CheckTouch, Switch

  Off(OUT_BC)

CheckSound:
  ReadSensor(IN_2, Volume)
  brcmp LT, CheckSound, Volume, 50

  RotateMotor(OUT_A,75,-90)

  OnRevReg(OUT_BC,100,OUT_REGMODE_SYNC)
  wait 200
  Off(OUT_BC)
  RotateMotor(OUT_C,75,-370)
  RotateMotor(OUT_B,75,370)
  OnFwdReg(OUT_BC,100,OUT_REGMODE_SYNC)

CheckLight:
  ReadSensor(IN_3, Level)
  brcmp GT, CheckLight, Level, 35

  Off(OUT_BC)

  RotateMotor(OUT_A,75,90)
endt
```

Note: This program may not work correctly – it causes my NXT to behave inconsistently.

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 additional information. We learned how to use the sound and the ultrasonic sensors, and how to use a motor as a rotation sensor. Finally, we wrote a program to do a fairly complex task using all the motors and sensors included with your NXT.

IX. Parallel threads

As has been indicated before, threads in NBC 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 thread drives the robot around in squares (like we did so often before) and the second thread checks for the touch sensor. When the sensor is touched, it moves a bit backwards, and makes a 90-degree turn.

```
#define MOVE_TIME   800
#define TURN_TIME   200

dseg segment
  Switch sword 0
  buf byte[]
dseg ends

thread main
  precedes move_square, check_sensors
  SetSensorTouch(IN_1)
  exit
endt

thread move_square
SquareLoop:
  OnFwd(OUT_BC,100)
  wait MOVE_TIME
  OnRev(OUT_C,100)
  wait TURN_TIME
  jmp SquareLoop
endt

thread check_sensors
SensorLoop:
  ReadSensor(IN_1,Switch)
  brtst EQ, SensorLoop, Switch
  OnRev(OUT_BC,100)
  wait 50
  Off(OUT_C)
  wait 85
  jmp SensorLoop
endt
```

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 threads may interfere. The following is happening. The robot is turning right, that is, the first thread is in its second sleep statement. Now the robot hits the sensor. It start going backwards, but at that very moment, the main thread is ready with sleeping and moves the robot forwards again; into the obstacle. The second thread 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 thread is sleeping we did not realize that the first thread was still running, and that its actions interfere with the actions of the second thread.

Using mutexes

A standard technique to solve this problem is to use a variable to indicate which thread is in control of the motors. The other threads are not allowed to drive the motors until the first thread indicates, using the variable, that it is ready. Such a variable is often called a mutex (for "mutually exclusive"). NBC provides a special variable type and functions for implementing mutexes. They are essentially Boolean (TRUE/FALSE) variables,

but can be easily manipulated using the `acquire` the `release` commands. Whenever a task wants to do something with the motors it first `acquires` the mutex, does what it needs, then `releases` the mutex. Because no thread can `acquire` the mutex if any other thread has it already (the acquire will sleep until the other thread `releases` the mutex), this ensures only one thread operates the motor at any time.

Here you find the program above, implemented using a mutex. When the touch sensor touches something, the mutex is acquired and the backup procedure is performed. During this procedure the task `move_square` must wait. At the moment the back-up is ready, the mutex is released and `move_square` can continue.

```
#define MOVE_TIME   800
#define TURN_TIME   200

dseg segment
  motor_control mutex
  Switch sword 0
  buf byte[]
dseg ends

thread main
  precedes move_square, check_sensors
  SetSensorTouch(IN_1)
  exit
endt

thread move_square
SquareLoop:
  acquire motor_control
  OnFwd(OUT_BC,100)
  release motor_control
  wait MOVE_TIME
  acquire motor_control
  OnRev(OUT_C,100)
  release motor_control
  wait TURN_TIME
  jmp SquareLoop
endt

thread check_sensors
SensorLoop:
  ReadSensor(IN_1,Switch)
  brtst EQ, SensorLoop, Switch
  acquire motor_control
  OnRev(OUT_BC,100)
  wait 50
  Off(OUT_C)
  wait 85
  release motor_control
  jmp SensorLoop
endt
```

Mutexes 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 learned how to use mutex variables to control the execution of tasks. This guarantees that at every moment only the critical part of one task is executed.

X. Communication between robots

To be completed.

Communication with other NXT bricks

Communication with a PC

Communication with other Bluetooth devices

Summary

XI. More commands

NBC has a number of additional commands. In this chapter we will discuss three types: the use of system calls, the system clock, arrays and type declarations.

System calls

The NXT firmware provides access to many low level functions via a `syscall` command. This command provides access to functions such as:

- NXT files (open, close, read, write, etc)
- Sounds (play tones, files, etc)
- Screen (text, polygons, bitmap graphics, etc)
- NXT buttons
- NXT system clock
- Random number generator
- Bluetooth communications
- Etc

In fact many of the macros you've already been using call this function to do their work. I won't go into any more detail in this tutorial, but I encourage you to read the NBC manual to find out what functions are available.

System clock

The NXT has a built in clock that allows you to time events down to millisecond accuracy. There are 2 ways to access this clock – syscall GetStartTick and gettick. While gettick returns the number of milliseconds since the NXT started, syscall GetStartTick returns the number of milliseconds since the current program started execution. In both cases you must pass an integer variable, which gets populated with the desired tick count.

The system clock is used in the `wait` command, but you can use it for anything you like. You can sleep for a particular amount of time by reading the clock 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.

```
#include "NXTDefs.h"

dseg segment
  cur_tick word
  end_tick word
  Switch sword
dseg ends

thread main
  SetSensorTouch(IN_1)
  gettick cur_tick
  add end_tick, cur_tick, 10000
  OnFwd(OUT_BC,100)

CheckSensor:
  ReadSensor(IN_1,Switch)
  brtst NEQ, StopIt, Switch
  gettick cur_tick
  brcmp LT, CheckSensor, cur_tick, end_tick

StopIt:
  Off(OUT_BC)
endt
```

Arrays

NBC allows you to declare and use arrays of any dimension, limited only by available memory. The following functions are provided for manipulating arrays:

- `arrinit arraydest, value, size`
  The arrinit opcode initializes the output array to the value (array, scalar, or user defined type) and size provided.
- `arrbuild arraydest, src1, src2, …, srcN`
  The arrbuild opcode constructs an output array from a variable number of input arrays, scalars, or aggregates.
- `index dest, arraysrc, index`
  The index opcode returns the arraysrc[index] value in the dest output argument.
- `replace arraydest, arraysrc, index, newval`
  The replace opcode replaces the arraysrc[index] item in arraydest with the newval input argument (arraysrc can be the same variable as arraydest to replace without copying the array; newval can be an array, in which case multiple items are replaced).
- `arrsize dest, arraysrc`
  The arrsize opcode returns the size of the input array in the scalar dest output argument.
- `arrsubset arraydest, arraysrc, index, length`
  The arrsubset opcode copies a subset of the input array to the output array.

Remember that arrays in NBC are always zero-based – the first element is element xero. Here is an example of how to manipulate arrays:

```
#include "NXTDefs.h"

dseg segment
  a byte[]
  b byte[]
  c byte[]
  aa byte[][]
  idx byte
  value byte
  tmp byte[]
dseg ends

thread main
  arrinit a, 0, 10 // a initialised to a[0..9]
  arrinit b, 1, 5 // b initialised to b[0..4]
  replace c, a, 0, b // c[0..4]=1, c[5-9]=0
  arrbuild aa, a, b, c // aa[0]=a, aa[1]=b, aa[2]=c
  arrsubset a, c, 3, 5 // a is now [1,1,0,0,0]

  arrsize idx, a // idx=5
  NumOut(0,0,1,idx)
  wait 1000

  arrsize idx, aa // idx=3
  NumOut(0,0,1,idx)
  wait 1000

  index value, a, 1 // value=1
  NumOut(0,0,1,value)
  wait 1000

  index value, a, 2 // value=0
  NumOut(0,0,1,value)
  wait 1000

  index tmp, aa, 2 // temp=[1,1,1,1,1,0,0,0,0,0]
  index value, tmp, 2 // value=1
  NumOut(0,0,1,value)
  wait 1000
endt
```

Type declarations

You can declare your own data structures in NBC, and use them to store related information. They are declared using the struct opcode, and like variables, must be contained in a data segment. Once a variable of user-defined type is defined, it's elements can be accessed using the familiar dot notation:

```
#include "NXTDefs.h"

dseg segment

TDate struct
  Day byte
  Month byte
  Year word
TDate ends

TEvent struct
  Date TDate
  Name byte[]
TEvent ends

  MyDate TDate
  MyEvent TEvent
  MyCalendar TEvent[]

dseg ends

thread main
  //Set date to 1st May 2006
  set MyDate.Day, 1
  set MyDate.Month, 5
  set MyDate.Year, 2006

  // Create an event
  mov MyEvent.Date, MyDate
  mov MyEvent.Name, 'May Day!'

  // Put the event in the calendar
  arrbuild MyCalendar, MyEvent
endt
```

Note that you can declare an array of user defined type, and your data structure can contain array elements.

Type aliases

You can define aliases for any built-in or user-defined type, using `typedef`. This can be useful if you have entities that can be described using a built-in type, but you want to give the entity its own data type. This may not seem all that useful, but it can make maintenance easier – if you need to change the data type for that entity, you only have to change it in 1 place, and all variables of that type will automatically change.

Here is a simple example:

```
#include "NXTDefs.h"

dseg segment
  small typedef byte // 8 bits
  medium typedef word // 16 bits
  big typedef dword // 32 bits

  MySmall small
  MyMedium medium
  MyBig big
dseg ends

thread main
  set MySmall, 65535
  add MySmall, MySmall, 65535
  set MyMedium, 65535
  add MyMedium, MyMedium, 65535
  set MyBig, 65535
  add MyBig, MyBig, 65535

  NumOut(0,20,0,MySmall) // displays 254
  NumOut(0,10,0,MyMedium) // displays 65534
  NumOut(0,0,0,MyBig) // displays 131070

  wait 1000
endt
```

Summary

In this chapter we found out about some more features of the NBC language. We looked at system calls, and the various things they can do, learned how to access the system clock, and finally, learned about how to declare arrays and user-defined types.

Final remarks

If you have worked your way through this tutorial you can now consider yourself an expert in NBC. 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 NBC 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 NBC program running in the NXT itself, using Bluetooth communication. Such a combination is very powerful.

You can find more information about the NXT from the Lego MindStorms web site.

> http://mindstorms.lego.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/
> http://forums.nxtasy.org/

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