Programming LEGO NXT Robots using NXC

(beta 30 or higher)

(Version 2.2, June 7, 2007)

by Daniele Benedettelli

with revisions by John Hansen

Preface

As happened for good old Mindstorms RIS, CyberMaster, and Spybotics, to unleash the full power of Mindstorms NXT brick, you need a programming environment that is more handy than NXT-G, the National Instruments Labview-like graphical language that comes with NXT retail set.

NXC is a programming language, invented by John Hansen, which was especially designed for the Lego robots. If you have never written a program before, don't worry. NXC is really easy to use and this tutorial will lead you on your first steps towards it.

To make writing programs even easier, there is the Bricx Command Center (BricxCC). This utility helps you to write your programs, to download them to the robot, to start and stop them, browse NXT flash memory, convert sound files for use with the brick, and much more. BricxCC works almost like a text processor, but with some extras. This tutorial will use BricxCC (version 3.3.7.16 or higher) as integrated development environment (IDE).

You can download it for free from the web at the address

http://bricxcc.sourceforge.net/

BricxCC runs on Windows PCs (95, 98, ME, NT, 2K, XP, Vista). The NXC language can also be used on other platforms. You can download it from the web page

http://bricxcc.sourceforge.net/nxc/

Most of this tutorial should also apply to other platforms, except that you loose some of the tools included in BricxCC and the color-coding.

The tutorial has been updated to work with beta 30 of NXC and higher versions. Some of the sample programs will not compile with versions older than beta 30.

As side note, my webpage is full of Lego Mindstorms RCX and NXT related content, including a PC tool to communicate with NXT:

http://daniele.benedettelli.com

Acknowledgements

Many thanks go to John Hansen, whose work is priceless!

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.

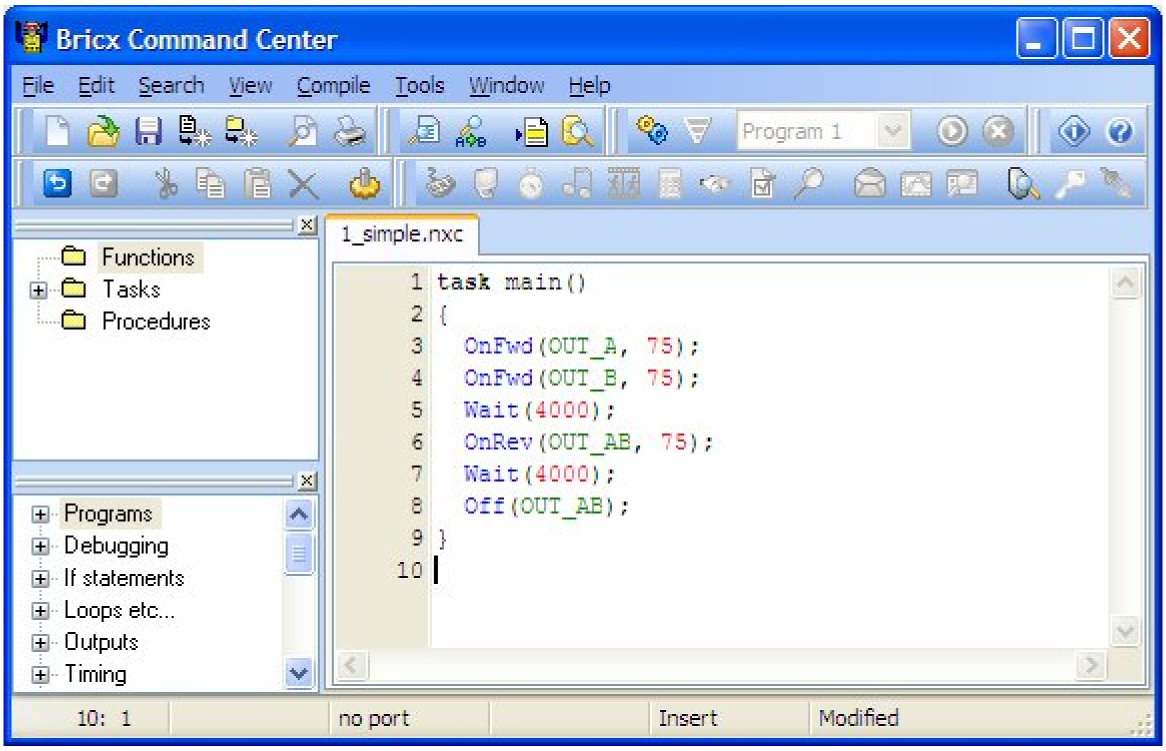
Building a robot

The robot we will use throughout this tutorial is Tribot, the first rover you have been instructed to build once got NXT set out of the box. The only difference is that you must connect right motor to port A, left motor to port C and the grabber motor to port B.

Make sure to have correctly installed Mindstorms NXT Fantom Drivers that come with your set.

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 BricxCC. 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 the text tab).



```
task main()
{
  OnFwd(OUT_A, 75);
  OnFwd(OUT_B, 75);
  Wait(4000);
  OnRev(OUT_AB, 75);
  Wait(4000);
  Off(OUT_AB);
}
```

The interface looks like a standard text editor, with the usual menu, 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, 75);
  OnFwd(OUT_C, 75);
  Wait(4000);
  OnRev(OUT_AC, 75);
  Wait(4000);
  Off(OUT_AC);
}
```

It might look a bit complicated at first, so let us analyze it.

Programs in NXC 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, 75);
```

This statement tells the robot to start output A, that is, the motor connected to the output labeled A on the NXT, to move forwards. The number following sets the speed of the motor to 75% of maximum speed.

```
OnFwd(OUT_C, 75);
```

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, that is, the number between the parentheses, gives the number of 1/1000 of a second: so you can very precisely tell the program how long to wait. For 4 seconds, the program is sleeping and the robot continues to move forwards.

```
OnRev(OUT_AC, 75);
```

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_AC` as argument. We could also have combined the first two statements this way.

```
Wait(4000);
```

Again we wait for 4 seconds.

```
Off(OUT_AC);
```

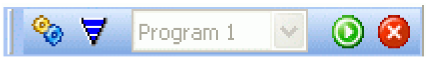
And finally we switch both motors off.

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 binary code that the robot can understand and execute) and sent to the robot using USB cable or BT dongle (called “downloading” the program).



Here you can see the button that allows you to (from left to right) compile, download, run and stop the program.

Press the second button and, assuming you made no errors when typing in the program, it will be correctly compiled and downloaded. (If there are errors in your program you will be notified; see below.)

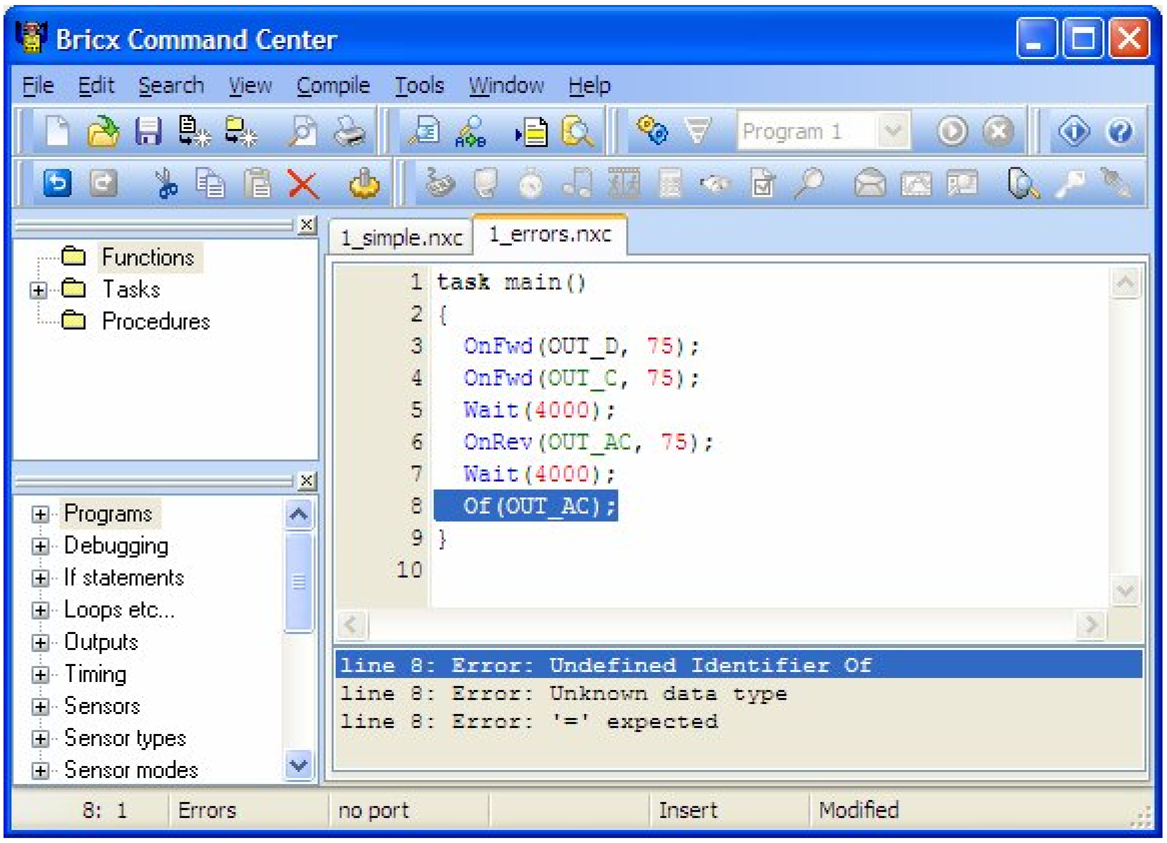
Now you can run your program. To this, go to My Files OnBrick menu, Software files, and run `1_simple` program. Remember: software files in NXT filesystem have the same name as source NXC file.

Also, to get the program run automatically, you can use the shortcut CTRL+F5 or after downloading the program you can press the green run button.

Does the robot do what you expected? If not, check wire connections.

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:



```
task main()
{
  OnFwd(OUT_D, 75);
  OnFwd(OUT_C, 75);
  Wait(4000);
  OnRev(OUT_AC, 75);
  Wait(4000);
  Of(OUT_AC);
}
```

```
line 8: Error: Undefined Identifier Of
line 8: Error: Unknown data type
line 8: Error: '=' expected
```

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 cause the wrong motor to turn. 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. To change the speed you just change the second parameter inside parentheses. The power is a number between 0 and 100. 100 is the fastest, 0 means stop (NXT servo motors will hold position). Here is a new version of our program in which the robot moves slowly:

```
task main()
{
  OnFwd(OUT_AC, 30);
  Wait(4000);
  OnRev(OUT_AC, 30);
  Wait(4000);
  Off(OUT_AC);
}
```

Summary

In this chapter you wrote your first program in NXC, using BricxCC. 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. BricxCC 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 NXC and only mention features of BricxCC when you really need them.

You also learned some important aspects of the language NXC. 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 basic motor commands: `OnFwd()`, `OnRev()` and `Off()`. Finally, you learned about the `Wait()` statement.

II. A more interesting program

Our first program was not so amazing. 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 NXC.

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_AC, 75);
  Wait(800);
  OnRev(OUT_C, 75);
  Wait(360);
  Off(OUT_AC);
}
```

You might have to try some slightly different numbers than 500 in the second `Wait()` command to make a 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 NXC you can define constant values as shown in the following program.

```
#define MOVE_TIME  1000
#define TURN_TIME   360

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(MOVE_TIME);
  OnRev(OUT_C, 75);
  Wait(TURN_TIME);
  Off(OUT_AC);
}
```

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 BricxCC 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    500
#define TURN_TIME    500

task main()
{
  repeat(4)
  {
    OnFwd(OUT_AC, 75);
    Wait(MOVE_TIME);
    OnRev(OUT_C, 75);
    Wait(TURN_TIME);
  }
  Off(OUT_AC);
}
```

The number inside the **`repeat`** statement's parentheses indicates how many times the code inside its brackets must be repeated. 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   1000
#define TURN_TIME    500

task main()
{
  repeat(10)
  {
    repeat(4)
    {
      OnFwd(OUT_AC, 75);
      Wait(MOVE_TIME);
      OnRev(OUT_C, 75);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_AC);
}
```

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 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 comments

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 BricxCC. The full program could look as follows:

```
/*  10 SQUARES

This program make the robot run 10 squares

*/

#define MOVE_TIME   500      // Time for a straight move
#define TURN_TIME   360      // Time for turning 90 degrees

task main()
{
  repeat(10)                 // Make 10 squares
  {
    repeat(4)
    {
      OnFwd(OUT_AC, 75);
      Wait(MOVE_TIME);
      OnRev(OUT_C, 75);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_AC);         // 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 us 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 therefore cannot be changed. We need a variable instead. Variables can easily be defined in NXC. Here is the spiral program.

```
#define TURN_TIME   360

int move_time;              // define a variable

task main()
{
  move_time = 200;           // set the initial value
  repeat(50)
  {
    OnFwd(OUT_AC, 75);
    Wait(move_time);        // use the variable for sleeping
    OnRev(OUT_C, 75);
    Wait(TURN_TIME);
    move_time += 200;         // increase the variable
  }
  Off(OUT_AC);
}
```

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 word **`int`** stands for integer. Only integer numbers can be stored in it. In the second line we assign the value 200 to the variable. From this moment on, whenever you use the variable, its value will be 200. 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 by 200. So the first time the robot sleeps 200 ms, the second time 400 ms, the third time 600 ms, and so on.

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. The next example does not have any effect on your robot hardware, since we don't know how to use the NXT display yet!

```
int aaa;
int bbb,ccc;
int values[];

task main()
{
  aaa = 10;
  bbb = 20 * 5;
  ccc = bbb;
  ccc /= aaa;
  ccc -= 5;
  aaa = 10 * (ccc + 3); // aaa is now equal to 80
  ArrayInit(values, 0, 10); // allocate 10 elements = 0
  values[0] = aaa;
  values[1] = bbb;
  values[2] = aaa*bbb;
  values[3] = ccc;
}
```

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. The variable named `values` is an array, that is, a variable that contains more than a number: an array can be indexed with a number inside square brackets. In NXC integer arrays are declared so:

`int name[];`

Then, this line allocates 10 elements initializing them to 0.

`ArrayInit(values, 0, 10);`

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 NXC 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(600);
    turn_time = Random(400);
    OnFwd(OUT_AC, 75);
    Wait(move_time);
    OnRev(OUT_A, 75);
    Wait(turn_time);
  }
}
```

The program defines two variables, and then assigns random numbers to them. `Random(600)` means a random number between 0 and 600 (the maximum value is not included in the range of numbers returned). Each time you call Random the numbers will be different.

Note that we could avoid the use of the variables by writing directly e.g. `Wait(Random(600))`.

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 (or at least until you press the dark grey button on NXT). You will learn more about the while statement in Chapter IV.

Summary

In this chapter you learned about the use of variables and arrays. You can declare other data types than `int`: `short`, `long`, `byte`, `bool` and `string`.

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 that is either positive or negative. If the number is less than 0 we make a right turn; otherwise we make a left turn. Here is the program:

```
#define MOVE_TIME    500
#define TURN_TIME    360

task main()
{
  while(true)
  {
    OnFwd(OUT_AC, 75);
    Wait(MOVE_TIME);
    if (Random() >= 0)
    {
      OnRev(OUT_C, 75);
    }
    else
    {
      OnRev(OUT_A, 75);
    }
    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() >= 0`. This means that `Random()` must be greater-than or equal to 0 to make the condition true. 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 omit them 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(1000);
    turn_time = Random(1000);
    OnFwd(OUT_AC, 75);
    Wait(move_time);
    OnRev(OUT_C, 75);
    Wait(turn_time);
    total_time += move_time;
    total_time += turn_time;
  }
  while (total_time < 20000);
  Off(OUT_AC);
}
```

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.

V. Sensors

Off course you can connect sensors to NXT to make the robot react to external events. Before I can show how to do this, we must change the robot a bit by adding a touch sensor. As before, follow the Tribot instructions to build the front bumper.

Connect the touch 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:

```
task main()
{
  SetSensor(IN_1,SENSOR_TOUCH);
  OnFwd(OUT_AC, 75);
  until (SENSOR_1 == 1);
  Off(OUT_AC);
}
```

There are two important lines here. The first line of the program tells the robot what type of sensor we use. `IN_1` is the number of the input to which we connected the sensor. The other sensor inputs are called `IN_2`, `IN_3` and `IN_4`. `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

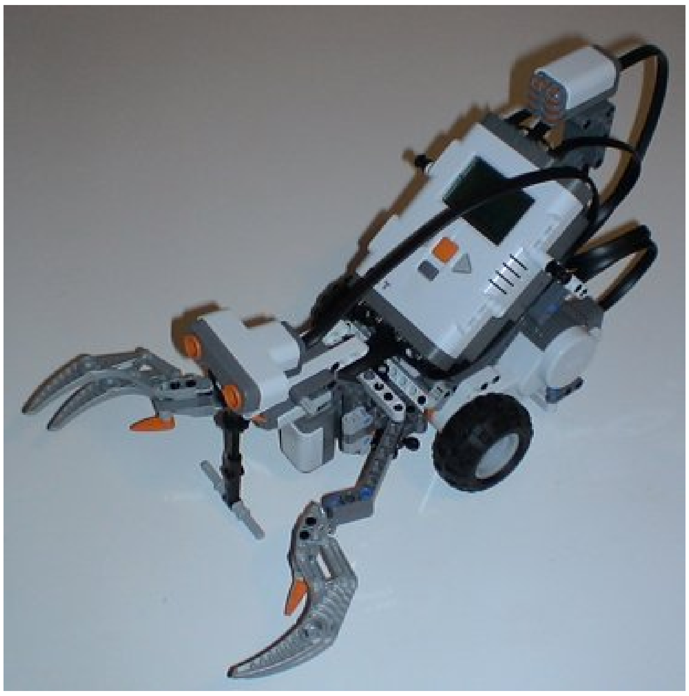
Let us now try to make the robot avoid obstacles. Whenever the robot hits an object, we let it move back a bit, make a turn, and then continue. Here is the program:

```
task main()
{
  SetSensorTouch(IN_1);
  OnFwd(OUT_AC, 75);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_AC, 75); Wait(300);
      OnFwd(OUT_A, 75); Wait(300);
      OnFwd(OUT_AC, 75);
    }
  }
}
```

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 300ms, turn right for 300ms, and then continue forwards again.

Light sensor

Besides the touch sensor, you also get a light sensor, a sound sensor and a digital ultrasonic sensor with Mindstorms NXT system. The light sensor can be triggered to emit light or not, so you can measure the amount of reflected light or ambient light in a particular direction. Measuring reflected light is particularly useful when making a robot follow a line on the floor. This is what we are going to do in the next example. To go on with experiments, finish building Tribot. Connect light sensor to input 3, sound sensor to input 2 and ultrasonic sensor to input 4, as indicated by instructions.



We also need the test pad with the black track that comes with the NXT set. The basic principle of line following is that the robot keeps trying to stay right on the border of the black line, turning away from line if the light level is too low (and sensor is in the middle of the line) and turning towards the line if the sensor is out of the track and detects a high light level. Here is a very simple program doing line following with a single light threshold value.

```
#define THRESHOLD 40

task main()
{
  SetSensorLight(IN_3);
  OnFwd(OUT_AC, 75);
  while (true)
  {
    if (Sensor(IN_3) > THRESHOLD)
    {
      OnRev(OUT_C, 75);
      Wait(100);
      until(Sensor(IN_3) <= THRESHOLD);
      OnFwd(OUT_AC, 75);
    }
  }
}
```

The program first configures port 3 as 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(100)` 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.

To read ambient light intensity with led off, configure sensor as follows:

```
SetSensorType(IN_3,IN_TYPE_LIGHT_INACTIVE);
SetSensorMode(IN_3,IN_MODE_PCTFULLSCALE);
ResetSensor(IN_3);
```

Sound sensor

Using the sound sensor you can transform your expensive NXT set into a clapper! We are going to write a program that waits for a loud sound, and drives the robot until another sound is detected. Attach the sound sensor to port 2, as described in Tribot instructions guide.

```
#define THRESHOLD 40
#define MIC SENSOR_2

task main()
{
  SetSensorSound(IN_2);
  while(true){
     until(MIC > THRESHOLD);
     OnFwd(OUT_AC, 75);
     Wait(300);
     until(MIC > THRESHOLD);
     Off(OUT_AC);
     Wait(300);
  }
}
```

We first define a `THRESHOLD` constant and an alias for `SENSOR_2`; in the main task, we configure the port 2 to read data from the sound sensor and we start a forever loop.

Using the **until** statement, the program waits for the sound level to be greater than the threshold we chose: note that `SENSOR_2` is not just a name, but a macro that returns the sound value read from the sensor.

If a loud sound occurs, the robot starts to go straight until another sound stops it.

The **wait** statements have been inserted because otherwise the robot would start and stop instantly: in fact, the NXT is so fast that takes no time to execute lines between the two **until** statements. If you try to comment out the first and the second **wait**, you will understand this better. An alternative to the use of **until** to wait for events is **while**, it is enough to put inside the parentheses a complementary condition, e.g.

```
while(MIC <= THRESHOLD).
```

There is not much more to know about NXT analog sensors; just remember that both light and sound sensors give you a reading from 0 to 100.

Ultrasonic sensor

Ultrasonic sensor works as a sonar: roughly speaking, it sends a burst of ultrasonic waves and measures the time needed for the waves to be reflected back by the object in sight. This is a digital sensor, meaning it has an embedded integrated device to analyze and send data. With this new sensor you can make a robot see and avoid an obstacle before actually hitting it (as is for touch sensor).

```
#define NEAR 15 //cm

task main(){
   SetSensorLowspeed(IN_4);
   while(true){
     OnFwd(OUT_AC,50);
     while(SensorUS(IN_4)>NEAR);
     Off(OUT_AC);
     OnRev(OUT_C,100);
     Wait(800);
   }
}
```

The program initializes port 4 to read data from digital US sensor; then runs forever a loop where robots goes straight until something nearer than NEAR cm (15cm in our example) is in sight, then backups a bit and begins going straight again.

Summary

In this chapter you have seen how to work with all sensors included in NXT set. We also saw how **until** and **while** commands are useful when using sensors.

I recommend you to write a number of programs by yourself at his point. You have all the ingredients to give your robots pretty complicated behavior now: try to translate in NXC the simplest programs shown in NXT retail software Robo Center programming guide.

VI. Tasks and subroutines

Up to now all our programs consisted of just one task. But NXC programs can have multiple tasks. It is also possible to put pieces of code in so-called subroutines that you can use in 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 NXC program consists of 255 tasks at most; each of them has a unique name. The task named `main` must always exist, since this is the first task to be executed. The other tasks will be executed only when a running task tells them to be executed or they are explicitly scheduled in the main; main task must terminate before they can start. From that moment on, both tasks are running simultaneously.

Let me show the use of tasks. 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 motors on and off in time) and watch for sensors. So it is better to use two tasks for this, one task that moves in squares; the other that reacts to the sensors. Here is the program.

```
mutex moveMutex;

task move_square()
{
  while (true)
  {
    Acquire(moveMutex);
    OnFwd(OUT_AC, 75); Wait(1000);
    OnRev(OUT_C, 75); Wait(500);
    Release(moveMutex);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      Acquire(moveMutex);
      OnRev(OUT_AC, 75); Wait(500);
      OnFwd(OUT_A, 75); Wait(500);
      Release(moveMutex);
    }
  }
}

task main()
{
  Precedes(move_square, check_sensors);
  SetSensorTouch(IN_1);
}
```

The main task just sets the sensor type and then starts both other tasks, adding them in the scheduler queue; after this, main task ends. Task `move_square` moves the robot forever in squares. Task `check_sensors` checks whether the touch sensor is pushed and, if so, drives the robot away from obstacle.

It is very important to remember that started tasks are running at the same moment and this can lead to unexpected results, if both tasks are trying to move motors as they are meant to do.

To avoid these problems, we declared a strange type of variable, **mutex** (that stands for **mut**ual **ex**clusion): we can act on this kind of variables only with the **Acquire** and **Release** functions, writing critical pieces of code between these functions, assuring that only one task at a time can have total control on motors.

These mutex-type variables are called semaphores and this programming technique is named concurrent programming; this argument is described on detail in chapter X.

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. Let us look at an example.

```
sub turn_around(int pwr)
{
  OnRev(OUT_C, pwr); Wait(900);
  OnFwd(OUT_AC, pwr);
}

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(1000);
  turn_around(75);
  Wait(2000);
  turn_around(75);
  Wait(1000);
  turn_around(75);
  Off(OUT_AC);
}
```

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 its name and passing a numerical argument writing it inside following parentheses. If a subroutine accepts no arguments, simply add parentheses with nothing inside them.

So it looks the same as many of the commands we have seen.

The main benefit of subroutines is that they are stored only once in the NXT and this saves memory. But when subroutines are short, it may be better to use **inline** functions instead. These are not stored separately but copied at each place they are used. This uses more memory but there is no limit on the number of inline functions. They can be declared as follows:

```
inline int Name( Args ) {
   //body;
   return x*y;
}
```

Defining and calling inline functions goes exactly the same way as with subroutines. So the above example, using inline functions, looks as follows:

```
inline void turn_around()
{
  OnRev(OUT_C, 75); Wait(900);
  OnFwd(OUT_AC, 75);
}

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(1000);
  turn_around();
  Wait(2000);
  turn_around();
  Wait(1000);
  turn_around();
  Off(OUT_AC);
}
```

In the above example, we can make the time to turn an argument of the function, as in the following examples:

```
inline void turn_around(int pwr, int turntime)
{
  OnRev(OUT_C, pwr);
  Wait(turntime);
  OnFwd(OUT_AC, pwr);
}

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(1000);
  turn_around(75, 2000);
  Wait(2000);
  turn_around(75, 500);
  Wait(1000);
  turn_around(75, 3000);
  Off(OUT_AC);
}
```

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. Note that in NXC, **sub** is the same as **void**; Also, functions can have other return type than **void,** can also return integer or string values to the caller: for details, see the NXC guide.

Defining macros

There is yet another way to give small pieces of code a name. You can define macros in NXC (not to be confused with the macros in BricxCC). 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_B, 75); Wait(3400);OnFwd(OUT_AB, 75);

task main()
{
  OnFwd(OUT_AB, 75);
  Wait(1000);
  turn_around;
  Wait(2000);
  turn_around;
  Wait(1000);
  turn_around;
  Off(OUT_AB);
}
```

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)  \
  OnFwd(OUT_A, s);OnRev(OUT_B, s);Wait(t);
#define turn_left(s,t)   \
  OnRev(OUT_A, s);OnFwd(OUT_B, s);Wait(t);
#define forwards(s,t)    OnFwd(OUT_AB, s);Wait(t);
#define backwards(s,t)   OnRev(OUT_AB, s);Wait(t);

task main()
{
  backwards(50,10000);
  forwards(50,10000);
  turn_left(75,750);
  forwards(75,1000);
  backwards(75,2000);
  forwards(75,1000);
  turn_right(75,750);
  forwards(30,2000);
  Off(OUT_AB);
}
```

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 of 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 skills 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 NXT has a built-in speaker that can play tones and even sound files. 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 or talk while it runs around.

Playing sound files

BricxCC has a built-in utility to convert .wav files into .rso files accessible via menu Tools → Sound conversion.

Then you can store .rso sound files on NXT flash memory using another utility, the NXT memory browser (Tools → NXT explorer) and play them with the command

`PlayFileEx(filename, volume, loop?)`

Its arguments are sound filename, volume (a number from 0 to 4), and loop: this last argument is set to 1 (TRUE) if you want the file to be looped or 0 (FALSE) if you want to play it only once.

```
#define TIME 200
#define MAXVOL 7
#define MINVOL 1
#define MIDVOL 3
#define pause_4th Wait(TIME)
#define pause_8th Wait(TIME/2)
#define note_4th \
  PlayFileEx("! Click.rso",MIDVOL,FALSE); pause_4th
#define note_8th \
  PlayFileEx("! Click.rso",MAXVOL,FALSE); pause_8th

task main()
{
  PlayFileEx("! Startup.rso",MINVOL,FALSE);
  Wait(2000);
  note_4th;
  note_8th;
  note_8th;
  note_4th;
  note_4th;
  pause_4th;
  note_4th;
  note_4th;
  Wait(100);
}
```

This nice program first plays the startup tune you might know already; then uses the other standard click sound to play “Shave and a haircut” jingle that made Roger Rabbit go crazy! The macros are really useful in this case to simplify notation in the main task: try modifying the volume settings to add accents in the tune.

Playing music

To play a tone, you can use the command `PlayToneEx(frequency, duration, volume, loop?)`

It has four arguments. The first is the frequency in Hertz, the second the duration (in 1/1000 of a second, like in the wait command), and the last are volume a loop as before. `PlayTone(frequency, duration)` can also be used; in this case the volume is the one set by NXT menu, and loop is disabled.

Here is a table of useful frequencies:

Sound	**3**	**4**	**5**	**6**	**7**	**8**	**9**

B	247	494	988	1976	3951	7902	
A#	233	466	932	1865	3729	7458	
A	220	440	880	1760	3520	7040	14080
G#		415	831	1661	3322	6644	13288
G		392	784	1568	3136	6272	12544
F#		370	740	1480	2960	5920	11840
F		349	698	1397	2794	5588	11176
E		330	659	1319	2637	5274	10548
D#		311	622	1245	2489	4978	9956
D		294	587	1175	2349	4699	9398
C#		277	554	1109	2217	4435	8870
C		262	523	1047	2093	4186	8372

As with the case of `PlayFileEx`, the NXT does not wait for the note to finish. So if you use multiple tones in a row then you had better add (slightly longer) wait commands in between. Here is an example:

```
#define VOL 3

task main()
{
  PlayToneEx(262,400,VOL,FALSE);  Wait(500);
  PlayToneEx(294,400,VOL,FALSE);  Wait(500);
  PlayToneEx(330,400,VOL,FALSE);  Wait(500);
  PlayToneEx(294,400,VOL,FALSE);  Wait(500);
  PlayToneEx(262,1600,VOL,FALSE); Wait(2000);

}
```

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

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

```
task music()
{
  while (true)
  {
    PlayTone(262,400);  Wait(500);
    PlayTone(294,400);  Wait(500);
    PlayTone(330,400);  Wait(500);
    PlayTone(294,400);  Wait(500);
  }
}

task movement()
{
  while(true)
  {
    OnFwd(OUT_AC, 75); Wait(3000);
    OnRev(OUT_AC, 75); Wait(3000);
  }
}

task main()
{
  Precedes(music, movement);
}
```

Summary

In this chapter you learned how to let the NXT play 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: ResetTachoCount, Coast (Float), OnFwdReg, OnRevReg, OnFwdSync, OnRevSync, RotateMotor, RotateMotorEx, and basic PID concepts.

Stopping gently

When you use the `Off()` command, the servo motor stops immediately, braking the shaft and holding position. It is also possible to stop the motors in a more gentle way, not using the brake. For this use `Float()` or `Coast()`command indifferently, that simple cut the power flowing to motor. 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_AC, 75);
  Wait(500);
  Off(OUT_AC);
  Wait(1000);
  OnFwd(OUT_AC, 75);
  Wait(500);
  Float(OUT_AC);
}
```

Advanced commands

The commands `OnFwd()` and `OnRev()` are the simplest routines to move motors.

The NXT servomotors have a built-in encoder that allows you to control precisely shaft position and speed;

NXT firmware implements a PID (Proportional Integrative Derivative) closed-loop controller to control motors' position and speed using encoders as feedback.

If you want your robot to go perfectly straight, you can use a **synchronization** feature that makes the selected **couple** of motors run together and wait for each other in case one of them is slowed down or even blocked; in a similar way, you can set a couple of motors to run together in sync, with a percentage of steering to turn left, right or spin in place, but always keeping sync. There are many commands to unleash servomotors' full power!

`OnFwdReg('ports','speed','regmode')` drives the motors specified by 'ports' at the 'speed' power applying the regulation mode that can be either `OUT_REGMODE_IDLE`, `OUT_REGMODE_SPEED` or `OUT_REGMODE_SYNC`. If IDLE is selected, no PID regulation will be applied; if SPEED mode is selected, the NXT regulates single motor to get a constant speed, even if load on motor varies; finally, if SYNC is selected, the couple of motors specified by 'ports' move in sync as explained before.

`OnRevReg()` acts as the precedent command, reversing direction.

```
task main()
{
  OnFwdReg(OUT_AC,50,OUT_REGMODE_IDLE);
  Wait(2000);
  Off(OUT_AC);
  PlayTone(4000,50);
  Wait(1000);
  ResetTachoCount(OUT_AC);
  OnFwdReg(OUT_AC,50,OUT_REGMODE_SPEED);
  Wait(2000);
  Off(OUT_AC);
  PlayTone(4000,50);
  Wait(1000);
  OnFwdReg(OUT_AC,50,OUT_REGMODE_SYNC);
  Wait(2000);
  Off(OUT_AC);
}
```

This program shows well different regulation if you try to stop wheels holding the robot in your hand: first (IDLE mode), stopping a wheel you will not notice anything; then (SPEED MODE), trying to slow down a wheel, you'll see that NXT increases motor's power to overcome your hold, trying to keep speed constant; finally (SYNC mode), stopping a wheel will cause the other one to stop, waiting for the blocked one.

`OnFwdSync('ports','speed','turnpct')` is the same as `OnFwdReg()` command in SYNC mode, but now you can also specify the 'turnpct' steering percentual (from -100 to 100).

`OnRevSync()` is the same as before, simply reversing the motor's direction. The following program shows these commands: try changing steering number to see how it behaves.

```
task main()
{
  PlayTone(5000,30);
  OnFwdSync(OUT_AC,50,0);
  Wait(1000);
  PlayTone(5000,30);
  OnFwdSync(OUT_AC,50,20);
  Wait(1000);
  PlayTone(5000,30);
  OnFwdSync(OUT_AC,50,-40);
  Wait(1000);
  PlayTone(5000,30);
  OnRevSync(OUT_AC,50,90);
  Wait(1000);
  Off(OUT_AC);
}
```

Finally, motors can be set to turn by a limited number of degrees (remember that a full turn is 360°).

For both following commands, you can act on motor's direction changing either the sign of the speed or the sign of the angle: so, if speed and angle have the same sign, motor will run forwards, if their sign is opposite, the motor will run backwards.

`RotateMotor('ports','speed','degrees')` rotates the motor shaft specified by 'ports' by a 'degrees' angle at 'speed' power (in 0-100 range).

```
task main()
{
  RotateMotor(OUT_AC, 50,360);
  RotateMotor(OUT_C, 50,-360);
}
```

`RotateMotorEx('ports','speed','degrees','turnpct','sync', 'stop')` is an extension of the precedent command, that lets you synchronize two motors (e.g. `OUT_AC`) specifying a `'turnpct'` steering percentage (from -100 to 100) and a boolean flag `'sync'` (that can be set to **true** or **false**). It also lets you specify whether the motors should brake after the angle of rotation has completed using the boolean flag 'stop'.

```
task main()
{
  RotateMotorEx(OUT_AC, 50, 360, 0, true, true);
  RotateMotorEx(OUT_AC, 50, 360, 40, true, true);
  RotateMotorEx(OUT_AC, 50, 360, -40, true, true);
  RotateMotorEx(OUT_AC, 50, 360, 100, true, true);
}
```

PID control

NXT firmware implements a digital PID (proportional integrative derivative) controller to regulate servomotors' position and speed with precision. This controller type is one of the simplest yet most effective closed loop feedback controller known is automation, and is often used.

In rough words, it works so (I'll talk about position regulation for a discrete time controller):

Your program gives the controller a set point R(t) to reach; it actuates the motor with a command U(t) , measuring its position Y(t) with the built-in encoder and calculates an error E(t) = R(t) – Y(t): here's why it is called a "closed loop controller", because the output position Y(t) is brought back to the controller's input to calculate the error. The controller transforms the error E(t) into the command U(t) so:

$U(t) = P(t) + I(t) + D(t)$, where

$P(t) = K_P \cdot E(t)$,

$I(t) = K_I \cdot (I(t-1) + E(t))$

and $D(t) = K_D \cdot (E(t) - E(t-1))$.

It could seem quite hard for a novice, but I'll try to explain this mechanism as best as I can.

The command is the sum of three contributes, the proportional part P(t), the integrative part I(t) and the derivative part D(t) .

P(t) makes the controller quick in time, but it does not assure a null error at equilibrium;

I(t) gives "memory" to the controller, in the sense that it takes trace of accumulated errors and compensates them, with the guarantee of a zero error at equilibrium;

D(t) gives "future prediction" to the controller (as derivation in math), speeding up response.

I know this can still be confusing, consider that entire academic books have been written on this argument! But we can still try it online, with our NXT brick! The simple program to fix things into memory is the following.

```
#define P 50
#define I 50
#define D 50

task main(){
   RotateMotorPID(OUT_A, 100, 180, P, I, D);
   Wait(3000);
}
```

The RotateMotorPID(port,speed, angle, Pgain,Igain,Dgain) let you move a motor setting different PID gains from the default ones. Try setting the following values

(50,0,0): the motor does not rotate 180° exactly, since an uncompensated error remains

(0,x,x): without proportional part, the error is very big

(40,40,0): there's an overshoot, that means the motor shaft moves beyond the set point and then turns back

(40,40,90): good precision and raising time (time to reach the set point)

(40,40,200): the shaft oscillate, since derivative gain is too high

Try other values to discover how these gains influence a motor's performance.

Summary

In this chapter you learned about the advanced motor commands available: `Float()`,`Coast()` that stop the motor gently; `OnXxxReg()`, and `OnXxxSync()` that allow feedback control on motors' speed and sync; `RotateMotor()` and `RotateMotorEx()` are used to turn motor's shaft by a precise number of degrees. You learned something about PID control too; it has not been an exhaustive explanation, but maybe I have caused a bit of curiosity in you: search the web about it!

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 old compatible RCX sensors, attaching them to NXT using Lego converter cables.

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 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 many different types, but I will report the main ones: `SENSOR_TYPE_TOUCH`, which is the touch sensor, `SENSOR_TYPE_LIGHT_ACTIVE`, which is the light sensor (with led on), `SENSOR_TYPE_SOUND_DB`, which is the sound sensor, and `SENSOR_TYPE_LOWSPEED_9V`, which is the Ultrasonic sensor. Setting the type sensor is in particular important to indicate whether the sensor needs power (e.g. to light up led in the light sensor), or to indicate NXT that the sensor is digital and needs to be read via I^{2}C serial protocol. It is possible to use old RCX sensor with NXT: `SENSOR_TYPE_TEMPERATURE`, for the temperature sensor, `SENSOR_TYPE_LIGHT` for old light sensor, `SENSOR_TYPE_ROTATION` for RCX rotation sensor (this type will be discussed later).

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. For details, see the NXC Programming Guide.

The second sensor mode is `SENSOR_MODE_BOOL`. In this mode the value is 0 or 1. When the raw value is above 562 the value is 0, otherwise it is 1. `SENSOR_MODE_BOOL` is the default mode for a touch sensor, but can be used for other types, discarding analogic informations. 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. `SENSOR_MODE_PERCENT` is the default mode for a light sensor. `SENSOR_MODE_ROTATION` is used 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. Off course, when you are counting edges or pulses, you should be able to set the counter back to 0. For this you use the command `ClearSensor()`, that clears the counter for the indicated sensor.

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(IN_1, SENSOR_TYPE_TOUCH);
  SetSensorMode(IN_1, SENSOR_MODE_PULSE);
  while(true)
  {
    ClearSensor(IN_1);
    until (SENSOR_1 > 0);
    Wait(500);
    if (SENSOR_1 == 1) {Off(OUT_AC);}
    if (SENSOR_1 == 2) {OnFwd(OUT_AC, 75);}
  }
}
```

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.

The rotation sensor

The rotation sensor is a very useful type of sensor: it is an optical encoder, almost the same as the one built inside NXT servomotors. The rotation sensor contains a hole through which you can put an axle, whose relative angular position is measured. One full rotation of the axle counts 16 steps (or –16 if you rotate it the other way), that means a 22.5 degrees resolution, very rough respect to the 1-degree resolution of the servomotor. This old kind of rotation sensor can be still useful to monitor an axle without the need to waste a motor; also consider that using a motor as encoder requires a lot of torque to move it, while old rotation sensor is very easy to rotate.

If you need finer resolution than 16 steps per turn, you can always use gears to mechanically increase the number of ticks per turn.

Next example is inherited from old tutorial for RCX.

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(IN_1, SENSOR_ROTATION); ClearSensor(IN_1);
  SetSensor(IN_3, SENSOR_ROTATION); ClearSensor(IN_3);
  while (true)
  {
    if (SENSOR_1 < SENSOR_3)
      {OnFwd(OUT_A, 75); Float(OUT_C);}
    else if (SENSOR_1 > SENSOR_3)
      {OnFwd(OUT_C, 75); Float(OUT_A);}
    else
      {OnFwd(OUT_AC, 75);}
  }
}
```

The program first indicates that both sensors are rotation sensors, and resets the values to zero. Next it starts 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

A little disclaimer is needed at the top of this section! Due to the new structure of improved NXT sensors and 6-wires cables, it is not easy as before (as was for RCX) to connect more sensors to the same port. In my honest opinion, the only reliable (and easy to do) application would be to build a touch sensor analog multiplexer to use in combination with a converter cable. The alternative is a complex digital multiplexer that can manage I^2C communication with NXT, but this is not definitely an affordable solution for beginners.

The NXT has four inputs to connect sensors. 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 (RCX sensors only). 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:

```
mutex moveMutex;
int ttt,tt2;

task moverandom()
{
  while (true)
  {
    ttt = Random(500) + 40;
    tt2 = Random();
    Acquire(moveMutex);
    if (tt2 > 0)
      { OnRev(OUT_A, 75); OnFwd(OUT_C, 75); Wait(ttt); }
    else
      { OnRev(OUT_C, 75); OnFwd(OUT_A, 75); Wait(ttt); }
    ttt = Random(1500) + 50;
    OnFwd(OUT_AC, 75); Wait(ttt);
    Release(moveMutex);
  }
}

task submain()
{
  SetSensorType(IN_1, SENSOR_TYPE_LIGHT);
  SetSensorMode(IN_1, SENSOR_MODE_RAW);
  while (true)
  {
    if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
    {
      Acquire(moveMutex);
      OnRev(OUT_AC, 75); Wait(300);
      Release(moveMutex);
    }
  }
}

task main()
{
  Precedes(moverandom, submain);
}
```

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.

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 NXT. 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 NXC 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 check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_AC, 75);
      Wait(500);
      OnFwd(OUT_A, 75);
      Wait(850);
      OnFwd(OUT_C, 75);
    }
  }
}

task submain()
{
  while (true)
  {
    OnFwd(OUT_AC, 75); Wait(1000);
    OnRev(OUT_C, 75); Wait(500);
  }
}

task main()
{
  SetSensor(IN_1,SENSOR_TOUCH);
  Precedes(check_sensors, submain);
}
```

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.

Critical sections and mutex variables

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.

```
mutex moveMutex;

task move_square()
{
  while (true)
  {
    Acquire(moveMutex);
    OnFwd(OUT_AC, 75); Wait(1000);
    OnRev(OUT_C, 75); Wait(850);
    Release(moveMutex);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      Acquire(moveMutex);
      OnRev(OUT_AC, 75); Wait(500);
      OnFwd(OUT_A, 75); Wait(850);
      Release(moveMutex);
    }
  }
}

task main()
{
  SetSensor(IN_1,SENSOR_TOUCH);
  Precedes(check_sensors, move_square);
}
```

The crux is that both the `check_sensors` and `move_square` tasks can control motors only if no other task is using them: this is done using the **Acquire** statement that waits for the `moveMutex` mutual exclusion variable to be released before using motors. The **Acquire** command counterpart is the **Release** command, that frees the mutex variable so other tasks can use the critical resource, motors in our case. The code inside the acquire-release scope is called critical region: critical means that shared resources are used. In this way tasks cannot interfere with each other.

Using semaphores

There is a hand-made alternative to **mutex** variables that is the explicit implementation of the **Acquire** and **Release** commands.

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 (same as mutex). We assume that a value of 0 indicates that no task is steering the motors (resource is free). Now, whenever a task wants to do something with the motors it executes the following commands:

```
until (sem == 0);
sem = 1; //Acquire(sem);

// Do something with the motors
// critical region

sem = 0;  //Release(sem);
```

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 move_square()
{
  while (true)
  {
    until (sem == 0); sem = 1;
    OnFwd(OUT_AC, 75);
    sem = 0;
    Wait(1000);
    until (sem == 0); sem = 1;
    OnRev(OUT_C, 75);
    sem = 0;
    Wait(850);
  }
}

task submain()
{
  SetSensor(IN_1, SENSOR_TOUCH);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      until (sem == 0); sem = 1;
      OnRev(OUT_AC, 75); Wait(500);
      OnFwd(OUT_A, 75); Wait(850);
      sem = 0;
    }
  }
}

task main()
{
  sem = 0;
  Precedes(move_square, submain);
}
```

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 NXT this chapter is for you (though you can still communicate data to the PC, having a single NXT). Robots can communicate with each other via Bluetooth radio technology: you can have multiple robots collaborate (or fight with each other), and you can build a big complex robot using two NXTs, so that you can use six motors and eight sensors.

For good old RCX, it is simple: it sends an InfraRed message and all robots around receive it.

For NXT it's a whole different thing! First, you must connect two or more NXTs (or NXT to PC) with the onbrick Bluetooth menu; only then you can send messages to connected devices.

The NXT that starts the connection is called Master, and can have up to 3 Slave devices connected on lines 1,2,3; Slaves always see the Master connected on line 0. You can send messages to 10 mailboxes available.

Master – Slave messaging

Two programs will be shown, one for the master, one for the slave. These basic programs will teach you how a fast continuous stream of string messages can be managed by a two-NXT wireless network.

The master program first checks if the slave is correctly connected on line 1 (`BT_CONN` constant) using `BluetoothStatus(conn)` function; then builds and sends messages with a `M` prefix and a growing number with `SendRemoteString(conn,queue,string)`, while receives messages from slave with `ReceiveRemoteString(queue,clear,string)` and displays data.

```
//MASTER
#define BT_CONN 1
#define INBOX 1
#define OUTBOX 5

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}

task main(){
   string in, out, iStr;
   int i = 0;
   BTCheck(BT_CONN); //check slave connection
   while(true){
     iStr = NumToStr(i);
     out = StrCat("M",iStr);
     TextOut(10,LCD_LINE1,"Master Test");
     TextOut(0,LCD_LINE2,"IN:");
     TextOut(0,LCD_LINE4,"OUT:");
     ReceiveRemoteString(INBOX, true, in);
     SendRemoteString(BT_CONN,OUTBOX,out);
     TextOut(10,LCD_LINE3,in);
     TextOut(10,LCD_LINE5,out);
     Wait(100);
     i++;
   }
}
```

The slave program is very similar, but uses `SendResponseString(queue,string)` instead of `SendRemoteString` because slave must can send messages only to its master, seen on line 0.

```
//SLAVE
#define BT_CONN 1
#define INBOX 5
#define OUTBOX 1

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}

task main(){
   string in, out, iStr;
   int i = 0;
   BTCheck(0); //check master connection
   while(true){
     iStr = NumToStr(i);
     out = StrCat("S",iStr);
     TextOut(10,LCD_LINE1,"Slave Test");
     TextOut(0,LCD_LINE2,"IN:");
     TextOut(0,LCD_LINE4,"OUT:");
     ReceiveRemoteString(INBOX, true, in);
     SendResponseString(OUTBOX,out);
     TextOut(10,LCD_LINE3,in);
     TextOut(10,LCD_LINE5,out);
     Wait(100);
     i++;
   }
}
```

You will notice that aborting one of the programs, the other will continue to send messages with growing numbers, without knowing that all the messages sent will be lost, because no one is listening on the other side. To avoid this problem, we could plan a finer protocol, with delivery acknowledgement.

Sending numbers with acknowledgement

Here we see another couple of programs: this time master sends numbers with `SendRemoteNumber(conn,queue,number)` and stops waiting for slave ack (`until` cycle, inside which we find `ReceiveRemoteString`); only if slave is listening and sending acks, the master proceeds sending the next message. Slave simply receives number with `ReceiveRemoteNumber(queue,clear,number)` and sends the ack with `SendResponseNumber`. Your master-slave programs must agree on the common code for the ack, in this case, I choose the hex value 0xFF.

The master sends random numbers and waits for slave ack; every time it receives an ack with the right code, the ack variable must be cleared, otherwise the master will continue sending without new acks, because the variable got dirty.

The slave checks continuously the mailbox and, if it is not empty, displays the read value and sends an ack to the master. At the beginning of the program, I choose to send an ack without reading messages to unblock the master; in fact, without this trick, if the master program is started for first, it would hang even if we start slave later. This way the first few messages get lost, but you can start master and slave programs in different moments without the risk of hanging.

```
//MASTER
#define BT_CONN 1
#define OUTBOX 5
#define INBOX 1
#define CLEARLINE(L)  \
  TextOut(0,L,"                   ");

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}

task main(){
   int ack;
   int i;
   BTCheck(BT_CONN);
   TextOut(10,LCD_LINE1,"Master sending");
   while(true){
     i = Random(512);
     CLEARLINE(LCD_LINE3);
     NumOut(5,LCD_LINE3,i);
     ack = 0;
     SendRemoteNumber(BT_CONN,OUTBOX,i);
     until(ack==0xFF) {
       until(ReceiveRemoteNumber(INBOX,true,ack) == NO_ERR);
     }
     Wait(250);
   }
}
```

```
//SLAVE
#define BT_CONN 1
#define OUT_MBOX 1
#define IN_MBOX 5

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}

task main(){
   int in;
   BTCheck(0);
   TextOut(5,LCD_LINE1,"Slave receiving");
   SendResponseNumber(OUT_MBOX,0xFF); //unblock master
   while(true){
     if (ReceiveRemoteNumber(IN_MBOX,true,in) != STAT_MSG_EMPTY_MAILBOX) {
       TextOut(0,LCD_LINE3,"                   ");
       NumOut(5,LCD_LINE3,in);
       SendResponseNumber(OUT_MBOX,0xFF);
     }
     Wait(10); //take breath (optional)
   }
}
```

Direct commands

There's another cool feature about Bluetooth communication: master can directly control its slaves.

In the next example, the master sends the slave direct commands to play sounds and move a motor; there is no need for a slave program, since it is the firmware of the slave NXT to receive and manage messages!

```
//MASTER
#define BT_CONN 1
#define MOTOR(p,s) RemoteSetOutputState(BT_CONN, p, s, \
  OUT_MODE_MOTORON+OUT_MODE_BRAKE+OUT_MODE_REGULATED, \
  OUT_REGMODE_SPEED, 0, OUT_RUNSTATE_RUNNING, 0)

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}

task main(){
   BTCheck(BT_CONN);
   RemotePlayTone(BT_CONN, 4000, 100);
   until(BluetoothStatus(BT_CONN)==NO_ERR);
   Wait(110);
   RemotePlaySoundFile(BT_CONN, "! Click.rso", false);
   until(BluetoothStatus(BT_CONN)==NO_ERR);
   //Wait(500);
   RemoteResetMotorPosition(BT_CONN,OUT_A,true);
   until(BluetoothStatus(BT_CONN)==NO_ERR);
   MOTOR(OUT_A,100);
   Wait(1000);
   MOTOR(OUT_A,0);
}
```

Summary

In this chapter we studied some of the basic aspects of Bluetooth communication between robots: connecting two NXTs, sending and receiving strings, numbers and waiting for delivery ackowledgments. This last aspect is very important when a secure communication protocol is needed.

As extra feature, you also learned how to send direct commands to a slave brick.

XII. More commands

NXC has a number of additional commands. In this chapter we will discuss three types: the use of the timer, commands to control the display, and the use of NXT file system.

Timers

The NXT has a timer that runs continuously. This timer ticks in increments of 1/1000 of a second. You can get the current value of the timer with `CurrentTick()`. Here is an example of the use of a timer. The following program lets the robot drive sort of random for 10 seconds.

```
task main()
{
  long t0, time;
  t0 = CurrentTick();
  do
  {
    time = CurrentTick()-t0;
    OnFwd(OUT_AC, 75);
    Wait(Random(1000));
    OnRev(OUT_C, 75);
    Wait(Random(1000));
  }
  while (time<10000);
  Off(OUT_AC);
}
```

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()
{
  long t3;
  SetSensor(IN_1,SENSOR_TOUCH);
  t3 = CurrentTick();
  OnFwd(OUT_AC, 75);
  until ((SENSOR_1 == 1) || ((CurrentTick()-t3) > 10000));
  Off(OUT_AC);
}
```

Don't forget that timers work in ticks of 1/1000 of a second just like the wait command.

Dot matrix display

NXT brick features a black and white dot matrix display with a resolution of 100x64 pixels. There are many API functions to draw text strings, numbers, dots, lines, rectangles, circles, and even bitmap images (.ric files). The next example tries to cover all these cases. Pixel numbered (0,0) is the bottom left one.

```
#define X_MAX 99
#define Y_MAX 63
#define X_MID (X_MAX+1)/2
#define Y_MID (Y_MAX+1)/2

task main(){
   int i = 1234;
   TextOut(15,LCD_LINE1,"Display", true);
   NumOut(60,LCD_LINE1, i);
   PointOut(1,Y_MAX-1);
   PointOut(X_MAX-1,Y_MAX-1);
   PointOut(1,1);
   PointOut(X_MAX-1,1);
   Wait(200);
   RectOut(5,5,90,50);
   Wait(200);
   LineOut(5,5,95,55);
   Wait(200);
   LineOut(5,55,95,5);
   Wait(200);
   CircleOut(X_MID,Y_MID-2,20);
   Wait(800);
   ClearScreen();
   GraphicOut(30,10,"faceclosed.ric");   Wait(500);
   ClearScreen();
   GraphicOut(30,10,"faceopen.ric");
   Wait(1000);
}
```

All these functions are quite self-explanatory, but now I'll describe their parameters in detail.

`ClearScreen()` clears the screen;

`NumOut(x, y, number)` lets you specify coordinates, and number;

`TextOut(x, y, string)` works as above, but outputs a text string

`GraphicOut(x, y, filename)` shows a bitmap **.ric** file

`CircleOut(x, y, radius)` outputs a circle specified by the coordinates of the center and radius;

`LineOut(x1, y1, x2, y2)` draws a line that goes from point `(x1,x2)` to `(x2,y2)`

`PointOut(x, y)` puts a dot on the screen

`RectOut(x, y, width, height)` draws a rectangle with the bottom left vertex in (x,y) and with the dimensions specified;

`ResetScreen()` resets the screen.

File system

The NXT can write and read files, stored inside its flash memory. So you could save a datalog from sensor data or read numbers during program execution. The only limit in files number and dimension is the size of the flash memory. NXT API functions let you manage files (create, rename, delete, find) , let you read and write text strings, numbers and single bytes.

In the next example, we will see how to create a file, write strings into it and rename it.

First, program deletes files with the names we're going to use: it is not a good habit (we should check for file existence, manually delete it or choose automatically another name for our work file), but there's no problem in our simple case. It creates our file by `CreateFile("Danny.txt", 512, fileHandle),` specifying name, size and a handle to the file, where NXT firmware will write a number for its own uses.

Then it builds strings and write to file with carriage return with `WriteLnString(fileHandle,string, bytesWritten),` where all the parameters must be variables. Finally, the file is closed and renamed. Remember: a file must be closed before beginning another operation, so if you created a file you can write to it; if you want to read from it, you must close it and open it with `OpenFileRead()`; to delete/rename it, you must close it.

```
#define OK LDR_SUCCESS

task main(){
   byte fileHandle;
   short fileSize;
   short bytesWritten;
   string read;
   string write;
   DeleteFile("Danny.txt");
   DeleteFile("DannySays.txt");
   CreateFile("Danny.txt", 512, fileHandle);
   for(int i=2; i<=10; i++ ){
      write = "NXT is cool ";
      string tmp = NumToStr(i);
      write = StrCat(write,tmp," times!");
      WriteLnString(fileHandle,write, bytesWritten);
   }
   CloseFile(fileHandle);
   RenameFile("Danny.txt","DannySays.txt");
}
```

To see the result, go to BricxCC→Tools→NXT Explorer, upload `DannySays.txt` to pc and take a look. Ready for the next example! We will create a table with ASCII characters.

```
task main(){
   byte handle;
   if (CreateFile("ASCII.txt", 2048, handle) == NO_ERR) {

   for (int i=0; i < 256; i++) {
      string s = NumToStr(i);
      int slen = StrLen(s);
      WriteBytes(handle, s, slen);
      WriteLn(handle, i);
    }
    CloseFile(handle);
  }
}
```

Really simple, this program creates the file and if no error occurred, it writes a number from 0 to 255 (converting it to string before) with `WriteBytes(handle, s, slen),` that is another way to write strings without carriage return; then it writes the number as is with `WriteLn(handle, value)` that appends a carriage return. The result, that you can see as before opening `ASCII.txt` with a text editor (as Windows Notepad), is so explainable: the number written as string is shown in a human-readable way, while the number written as hex value is interpreted and shown as an ASCII code.

Two important functions remain to be showed: `ReadLnString` to read strings from files and `ReadLn` to read numbers.

Now for the example for the first one: the main task calls `CreateRandomFile` subroutine that creates a file with random numbers in it (written as strings); you can comment this line and use another hand-created text file for this example.

Then the main task opens this file for reading, reads it a line at once until the end of file, calling `ReadLnString` function and displays text.

In the `CreateRandomFile` subroutine we generate a predefined quantity of random numbers, convert them to string and write them to the file.

The `ReadLnString` accepts a file handle and a string variable as arguments: after the call, the string will contain a text line and the function will return an error code, that we can use to know if the end of file has been reached.

```
#define FILE_LINES 10

sub CreateRandomFile(string fname, int lines){
   byte handle;
   string s;
   int bytesWritten;
   DeleteFile(fname);
   int fsize = lines*5;
   //create file with random data
   if(CreateFile(fname, fsize, handle) == NO_ERR) {
      int n;
      repeat(FILE_LINES) {
         int n = Random(0xFF);
         s = NumToStr(n);
         WriteLnString(handle,s,bytesWritten);
      }
      CloseFile(handle);
   }
}

task main(){
   byte handle;
   int fsize;
   string buf;
   bool eof = false;
   CreateRandomFile("rand.txt",FILE_LINES);
   if(OpenFileRead("rand.txt", fsize, handle) == NO_ERR) {
     TextOut(10,LCD_LINE2,"Filesize:");
     NumOut(65,LCD_LINE2,fsize);
     Wait(600);
     until (eof == true){   // read the text file till the end
         if(ReadLnString(handle,buf) != NO_ERR) eof = true;
         ClearScreen();
         TextOut(20,LCD_LINE3,buf);
         Wait(500);
     }
   }
   CloseFile(handle);
}
```

In the last program, I'll show you how to read numbers from a file.
I take the occasion to give you a little sample of conditional compilation. At the beginning of the code, there is a definition that is not used for a macro neither for an alias: we simply define INT.

Then there is a preprocessor statement

```
#ifdef INT
   …Code…
#endif
```

that simply tells the compiler to compile the code between the two statements if INT as been previously defined. So, if we define INT, the task main inside the first couplet will be compiled and if LONG is defined instead of INT, the second version of main will be compiled.

This method allows me to show in a single program how both `int` (16 bit) and `long` (32 bit) types can be read from file calling the same function `ReadLn(handle,val)`.

As before, it accepts a file handle and a numeric variable as arguments, returning an error code.

The function will read 2 bytes from file if the passed variable is declared as `int`, and will read 4 bytes if the variable is `long`. Also `bool` variables can be written and read the same way.

```
#define INT //  INT or LONG

#ifdef INT
task main () {
   byte handle, time = 0;
   int n, fsize,len, i;
   int in;
   DeleteFile("int.txt");
   CreateFile("int.txt",4096,handle);
   for (int i = 1000; i<=10000; i+=1000){
      WriteLn(handle,i);
   }
   CloseFile(handle);
   OpenFileRead("int.txt",fsize,handle);
   until (ReadLn(handle,in)!=NO_ERR){
      ClearScreen();
      NumOut(30,LCD_LINE5,in);
      Wait(500);
   }
   CloseFile(handle);
}
#endif

#ifdef LONG
task main () {
   byte handle, time = 0;
   int n, fsize,len, i;
   long in;
   DeleteFile("long.txt");
   CreateFile("long.txt",4096,handle);
   for (long i = 100000; i<=1000000; i+=50000){
      WriteLn(handle,i);
   }
   CloseFile(handle);
   OpenFileRead("long.txt",fsize,handle);
   until (ReadLn(handle,in)!=NO_ERR){
      ClearScreen();
      NumOut(30,LCD_LINE5,in);
      Wait(500);
   }
   CloseFile(handle);
}
#endif
```

Summary

In this last chapter you met the advanced features offered by NXT: high resolution timer, dot matrix display and filesystem.

XIII. Final remarks

If you have worked your way through this tutorial you can now consider quite expert in NXC. 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 unbelievable things.

This tutorial did not cover all aspects of the BricxCC. You are recommended to read the NXC Guide at every chapter. Also, NXC 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 drive a Lego robot directly from a PC. This requires you to write a program in a language like C++, Visual Basic, Java or Delphi. It is also possible to let such a program work together with an NXC program running in the NXT itself. Such a combination is very powerful. If you are interested in this way of programming your robot, best start with downloading the Fantom SDK and Open Source documents from the NXTreme section of Lego MindStorms web site.

http://mindstorms.lego.com/Overview/NXTreme.aspx

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/robotics/nxt