RobotC primer

version 1.4, September 2014

Alexander Kirillov

IslandBots robotic club

URL: `http://islandbots.org/`

E-mail address: `shurik179@gmail.com`

This work is based in part on *Programming with Robots* by Albert W. Schueller, available from http://carrot.whitman.edu/Robots/.

Contents

Introduction

These notes contain a short introduction to programming LEGO Mindstorms NXT robots using RobotC programming language. It is intended for people with little to none programming experience and minimal experience with LEGO NXT set.

It was originally written for *SigmaCamp* (`http://sigmacamp.org`), a science summer camp where the author was teaching a course on robotics, and for members of *IslandBots* robotics club (`http://islandbots.org`), of which the author is the coach.

If you have any comments, suggestions, or corrections, please send them to shurik179@gmail.com.

A latest version of this text is always available at `http://islandbots.org/robotc/`.

Chapter 1

"Hello world!" in RobotC

1.1. What is LEGO Mindstorms NXT?

LEGO Mindstorms NXT is a robotics construction set manufactured by LEGO. The heart of LEGO Mindstorms set is the NXT brick; it is a small computer equipped with a mini display and ports for connecting motors and sensors. The basic set contains 3 motors, a touch sensor, a sound sensor, ultrasonic sensor and a light sensor (in NXT 2.0, light sensor was replaced by the color sensor). A variety of third party sensors is also available.

In addition, the set contains large number of LEGO Technic building elements such as beams, pins, gears, and axles.

Most projects in our primer will be using a simple robot with two drive wheels powered by motors attached to ports B and C and a third wheel which can freely rotate (caster wheel) or a caster ball. Some good starting models are:

- The Castor Bot (`http://tinyurl.com/castorbot2`), from the site `http://nxtprograms.com`. This robot is shown in Figure 1.1 below.
- The tribot designed by LEGO: `http://tinyurl.com/o4yur4g`.
- The Robot Educator Model from the creators of RobotC: `http://tinyurl.com/pgwsqcv`

Choose one of the models above and build it — or create your own one robot. Make sure that you read the robot-building tips in Section 10.1.

1.2. RobotC

The robot can be programmed using either LEGO's graphical programming environment, NXT-G (which is quite intuitive but limited — writing a complex program in it is really frustrating) or using one of several alternative languages. In this book we will be using RobotC. This language is a variant of probably the most popular programming language ever, C, tailored specifically for robotics. It was developed by Carnegie Mellon University. RobotC can be downloaded from `http://www.robotc.net/`. It is not free: after the free trial period of one month, you need to buy a license. An annual license cost $49 - or you can buy a perpetual license for $79 (as of August 2014).

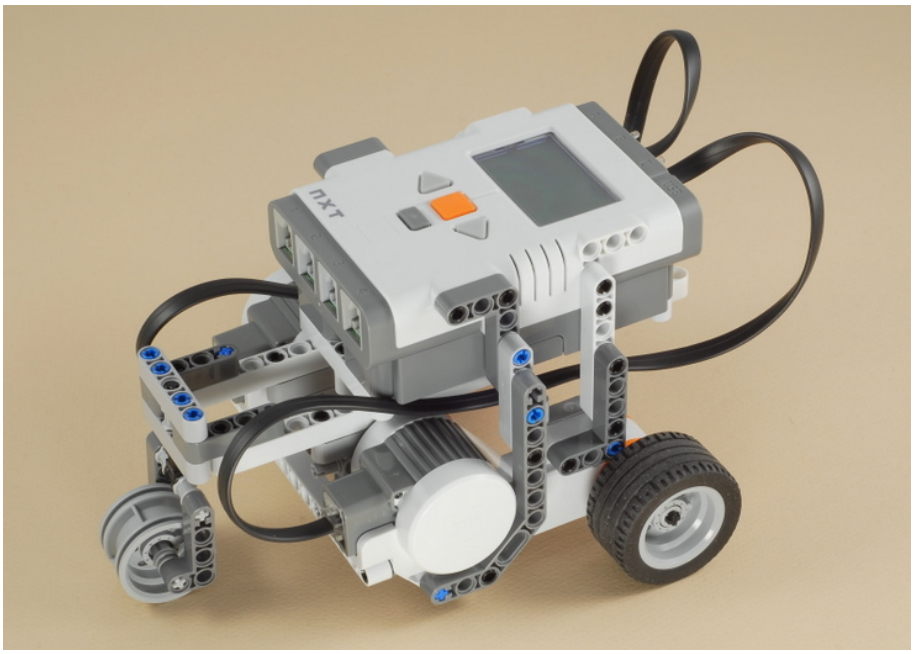

Figure 1.1. The Castor Bot, from nxtprograms.com

As of this writing, RobotC is available for Windows only — there is no Mac or Linux version. The procedure for downloading, installing and activating RobotC is well documented on RobotC website.

The latest (as of August 2014) version is RobotC 4.10, still in beta. This Primer uses RobotC version 3.62. All sample programs given in this primer should still work with RobotC 4.x.

The main window of RobotC (version 3) is shown in Figure 1.2.

1.3. Connecting to NXT

After installing RobotC, you need to establish communication between the computer and NXT brick. Note that RobotC requires installing a special firmware on the NXT, different from the factory installed one used by LEGO graphical environment.

To install the firmware, start RobotC, turn the NXT brick on and connect it to the computer by a USB cable; if you are doing it for the first time, you may see a Windows notification about installing NXT drivers (this should be done automatically). After this, go to the RobotC menu and select Robot→Download Firmware→Standard File. Note that the word “download” is somewhat misleading: you are not downloading the firmware from the internet (it is already on your computer – the firmware is included with RobotC); instead, you are downloading the firmware from your computer to the NXT brick.

This will produce a window like the one shown in Figure 1.3.

The section `NXT bricks currently connected via USB` should show your NXT brick. Click on it to select it and then click on the `F/W download` button. This begins the download process.

If you get the message `Firmware download completed`, then everything is good. To test your setup, go to Robot→NXT Brick→Communication Link Setup. In the top left corner, the

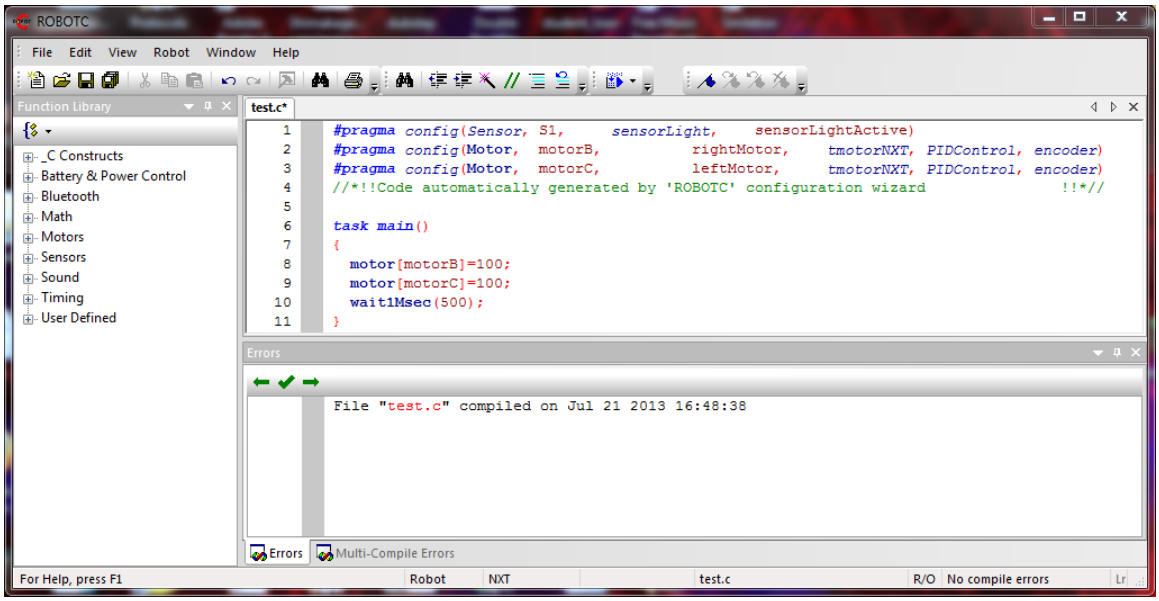

Figure 1.2. Main window of RobotC. The main part is where you write the program; below it, the area for compiler messages such as warnings and error notifications.

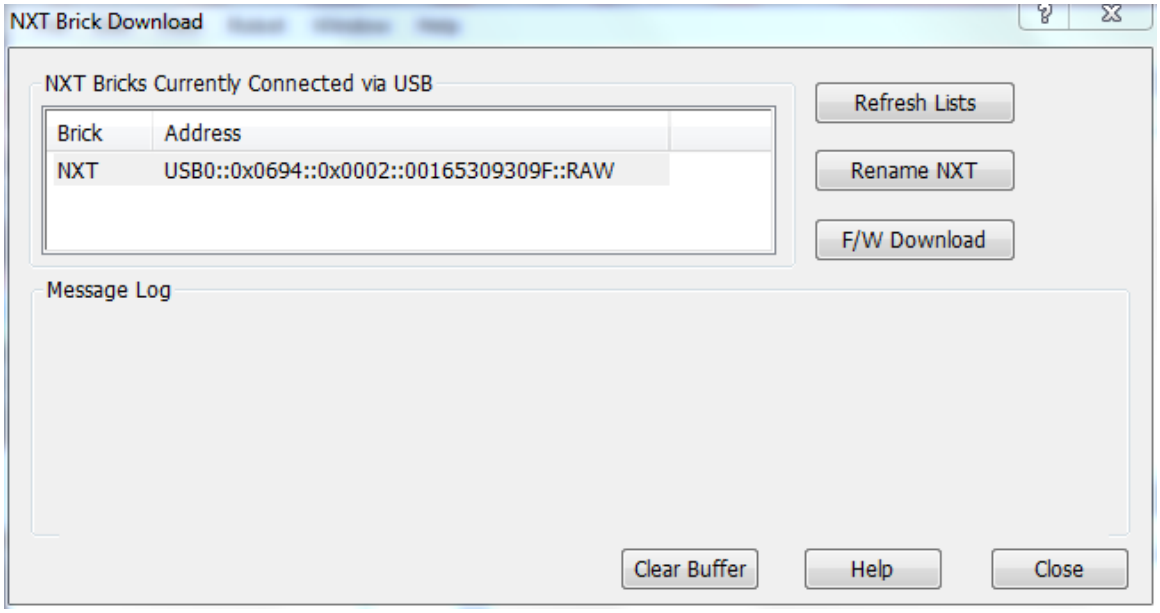

Figure 1.3. Installing the firmware

section `NXT bricks currently connected via USB` should show your NXT brick. Click on it to select it and then on `Select` button.

If for some reason the firmware failed to download and you are left with an NXT brick which does not even turn on and instead just makes some clicking noise — do not panic. This is a relatively common problem and can usually be fixed following steps in Section 10.2.

Note. *The firmware version must be matched to RobotC version, so if you update RobotC, you must install the new version of the firmware to the NXT brick (the new version of firmware is included*

with RobotC, so there is no need to download it from the internet separately). And if you later decide to use the brick with LEGO's graphical programming environment, you will need to reinstall LEGO's firmware onto NXT, using LEGO NXT software.

One can also connect NXT brick to the computer using Bluetooth, thus eliminating the need for USB cables. However, in my experience this works somewhat unreliably, so I recommend using USB if at all possible.

1.4. The first program

To create a new program, select from the menu File→New.

The basic structure of the program looks like this:

```
task main(){
  command;
  command;
  ...
}
```

where each command is an instruction for the robot. These instructions are executed one by one, until the program reaches the end; at this point, the program stops. Note that each command must be followed by a semicolon — if you forget one (which is probably the most common beginner's mistake), the program will not work.

The meaning of the words in the first line (`task main(){`) will be explained later; for now, just make sure you enter them exactly as written — and do not forget the matching curly bracket at the very end.

Note that in general, everything is case-sensitive: `main` is not the same as `Main`, so you should be careful when entering. On the other hand, spaces and line breaks are not important: you can put several commands on the same line, separate them with spaces, or put in empty lines to make it look good (more on that in Section 2.5).

The following is a simplest RobotC program — an analog of “Hello World!” in RobotC.

Note. *From now on, we use the following typographic convention: things which are just placeholders, to be replaced by actual commands (or names, or..) in the program, such as the word “command” in the listing above, will be shown in italics.*

```
// Go forward for 0.5 seconds
task main(){
  motor[motorB]=100;
  motor[motorC]=100;
  wait1Msec(500);
}
```

The first two commands tell the robot to turn on motors B and C (that is, motors plugged in the corresponding ports on NXT brick) at 100% power forward. The next command just tells the NXT brick to wait the specified number of milliseconds (in this case, 500 ms=0.5 sec) doing nothing new (but keeping whatever power was given to motors by the previous commands). As a result, the robot will go forward for 0.5 sec.

The line beginning with the double slash `//` is a *comment*; it is completely ignored by the computer. Comments are usually inserted to explain the working of the program to the human reader. Even in the simplest programs, inserting comments really improves readability of the code; in large programs, it is a must. RobotC treats anything following `//` as a comment (until the end of the line). You can also create multi-line comments: any text between `/*` and `*/` is treated as a comment:

```
/*This is an example
  of a multi-line comment
*/

task main(){
  motor[motorB]=100;
  motor[motorC]=100;
  wait1Msec(500);
}
```

1.5. Compiling and downloading the program

To download the program you created to the robot and run it, first save the program by selecting File→Save in the menu or using Ctrl-S shortcut. After this, connect the robot to the computer using a USB cable, make sure the NXT is turned on, and download the program using Robot→Compile and Download Program or F5 shortcut.

If there were no errors in the program, you will see a window titled "Program Debug" like the one shown below.

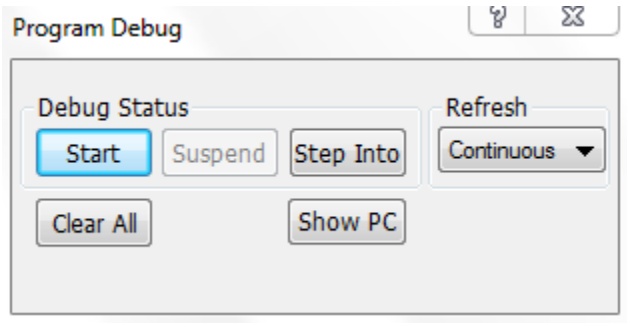

Figure 1.4. Program Debug window

You can now run the program by clicking on `Start` button in Program Debug window or by disconnecting the USB cable from the NXT and using the buttons on the NXT to select My Files→Software Files→Program name→Run. The second way is preferred: you do not want your robot to be moving while it is still connected to the computer by a USB cable.

If you did have some errors in the program, you will see error messages; the lines in your program that the computer was unable to process will be marked by red crosses. You will need to fix the errors before re-downloading the program.

If your first program worked correctly, you can now experiment by modifying it. For example, you can make the robot move backwards by replacing `motor[motorB]=100` by `motor[motorB]=-100`. You can also have different motors run at different power; this will force the robot to turn. For

example, setting power of one motor to 100 and the other to -100 will make the robot turn in place. To stop the motors, give each of them power of 0.

Using this, try writing a program for the robot to make a perfect square.

1.6. Getting help with RobotC

RobotC comes with several great sources of information that you can use (in addition to this primer):

- RobotC help, available in the menu under Help, provides detailed documentation of all of RobotC functions and structures. It is also available online at `http://help.robotc.net/WebHelpMindstorms/index.htm` (note that the online version maybe slightly outdated).
- RobotC includes a large number of sample programs, which allow one to learn by example. These programs are available under File→Open Sample Program. Note that you will not be able to modify these programs (the files containing them are write-protected), but you can copy them to your own files and experiment there.
- A number of very detailed tutorials and guides can be found on RobotC website at `http://www.robotc.net/education/curriculum/nxt/`.

Chapter 2

Loops and ifs

In the previous section we said that the simplest program consists of a series of commands which are executed one by one. However, this is clearly insufficient: for any serious work, we need more tools — for example allowing the program to repeat a given sequence of commands several times. Such tools will be introduced in this chapter.

2.1. *If* statements and first use of sensors

The most basic control structure is called an `if` statement. Its syntax is as follows:

```
if (condition) {
  some commands
} else {
  some other commands
}
```

Here `condition` must be some condition — a statement that can either be true or false; exact syntax of conditionals will be discussed in Section 4.4. Depending on whether this condition is true or not, the program will execute either the first sequence of commands or the second one.

The `else` part can be omitted:

```
if (condition) {
  some commands
}
```

In this case, the program will check the condition; if it is true, the commands inside the curly braces will be executed; if not, no commands will be executed.

Please pay attention to the syntax: each sequence of commands must be enclosed in curly braces and the condition must be enclosed in parentheses. In general, curly braces are used to group commands together. Note also that you should not put a semicolon after the closing curly brace, but each command inside the braces (including the last one) must be followed by a semicolon.

To illustrate `if` statements, let us make our robot check the color of the floor and react to it. To do that, we attach a light sensor to port 1 of the robot. The light sensor should be pointing down at the floor and should be about 1–1.5cm (about 1/2 inch) off the floor.

The following program would read the values of the light sensor and depending on the value move the robot either forward or backwards:

```
#pragma config(Sensor, S1, lightSensor, sensorLightActive)
task main(){
  if (sensorValue[lightSensor]>40) { // robot is on white
    motor[motorB]=100;
    motor[motorC]=100;
    wait1Msec(500);
  } else {  // robot is on black
    motor[motorB]=-100;
    motor[motorC]=-100;
    wait1Msec(500);
  }
}
```

The first line of this program sets up the sensor, telling RobotC what kind of sensor it is (light sensor, generating light), into what port it is plugged in (S1 means port 1), and what name we will be using for this sensor (lightSensor). We will discuss how one sets up sensors in more detail in Section 6.1; for now, just enter this line as shown. If you are using the NXT 2.0 set which instead of a light sensor has a color sensor, use the color sensor and replace the first line with

```
#pragma config(Sensor, S1, lightSensor, sensorCOLORRED)
```

The condition `sensorValue[lightSensor]>40` checks whether the current sensor reading is greater than 40. The sensor reading varies from 0–100; experience shows that typical readings are about 25 when robot is on a black surface, and about 50 when robot is on a white sheet of paper. Thus, the condition `sensorValue[lightSensor]>40` will be true when the robot is on white, and false when it is on black. In the first case, the robot will go forward for 0.5 sec; in the second, it will go backward.

2.2. *While* loops

Another common programming construction is called a loop, which allows one to repeat a group of commands more than once. The simplest kind of loop is the `while` loop, which has the following syntax:

```
while (condition) {
  some commands
}
```

This snippet of code would check whether condition is true. If it is, it will execute the group of commands in braces (usually called *body of the loop*) and then return to the beginning of the loop to check the condition again. This repeats while the condition is true; if at the next check the condition fails, the program skips the loop body and moves on to the commands following the loop.

Warning. Note that the condition is checked only in the beginning of each cycle. It is not checked between the commands in the loop.

Here is the simplest example of the while loop in action. As in the previous section, we use a light sensor attached to port 1.

```
#pragma config(Sensor, S1, lightSensor, sensorLightActive)
task main(){
while (sensorValue[lightSensor]<40) {
  motor[motorB]=100;
  motor[motorC]=100;
}
motor[motorB]=0;
motor[motorC]=0;
}
```

This program makes the robot go forward while the sensor shows values of less than 40 (that is, while the robot is on black surface); as soon as this fails (the robot reaches white), it stops the motors. In other words, this program will make the robot go forward until it sees white.

It can be improved. Namely, it is not necessary to set the motor power to 100 at every execution of the loop: if the motor is already running at 100% power, why set the power to 100 again? We could just leave the motor alone and do nothing. This gives the following program:

```
#pragma config(Sensor, S1, lightSensor, sensorLightActive)
task main(){
  motor[motorB]=100;
  motor[motorC]=100;
  while (sensorValue[lightSensor]<40) { }
  motor[motorB]=0;
  motor[motorC]=0;
}
```

As you can see, in this case the body of the loop is empty. Yes, this is allowed.

2.3. Repeating indefinitely: *while(true)*

It is common to have the robot repeat the same task indefinitely (or, realistically, until you physically turn the robot off). A typical example would be a robotic vacuum that goes around the room.

The easiest way to do that in RobotC is to use a `while` loop with a condition that is always true. One could use something like `while (1>0)`, but there is a more straightforward alternative: RobotC provides a special notation for a logical expression which is always true. Not surprisingly, it is denoted simply by `true`, so one can write

```
while (true) {
  commands to be repeated indefinitely
}
```

2.4. *For* loops and switches

RobotC also provides another kind of a loop, in which the loop body is repeated a specified number of times. This is called a `for` loop. However, it is much less useful in robot programming, so we do not discuss it here.

Another commonly used control structure is called a `switch`: it allows the program to execute one of several command blocks depending on the value of an expression. Unlike the `if` statement which only allows for two alternatives, the number of possible values for `switch` is not limited.

Detailed description of both the `for` loop and the `switch` statement can be found in RobotC documentation.

2.5. Code formatting

As was mentioned before, the computer completely disregards spaces and line breaks, so the example illustrating the `if` statement in Section 2.1 could also be written as follows:

```
#pragma config(Sensor, S1, lightSensor, sensorLightActive)
task main(){
if (SensorValue[lightSensor]>40) { motor[motorB]=100;
motor[motorC]=100;  wait1Msec(500);} else {motor[motorB]=-100;
motor[motorC]=-100;
wait1Msec(500);}
}
```

It is clear, however, that in this form the program is much more difficult to read and understand (for a human), and much easier to make a mistake. Therefore, it is a common practice among programmers to format the code so that it is easy to see where a loop or another structure begins and ends:

- The body of the loop is indented (usually by two spaces, but it is a matter of personal preference)
- The opening curly brace is placed on the same line as the `if` or `while` statement itself, and the closing curly brace is on a separate line, aligned with the statement
- If a loop is long, a comment is added to indicate where a loop ends

```
while (true) {
  if (x>y) {
    command;
    command;
  } else {
    command;
  }
} //end of while loop
```

Chapter 3

Project: Line Follower

In this chapter, we do our first project: a robot that follows a line on the floor. We will make a line by putting 1-inch wide white masking tape on a black surface (a sheet of plywood painted black). You can make your own field; just make sure the line is at least one inch wide and doesn't have sharp turns. Our robot will be using one light sensor to detect the line; the sensor will be attached to the front of the robot.

Before we start writing code, we need to describe the algorithm the robot will be using — first in human language, then translate it to RobotC.

The obvious algorithm is "start on the line; go forward until you get off the line (that is, until the sensor sees black); turn to get back on the line; repeat". The problem with this algorithm is in getting back on the line. The light sensor can tell us that we are off the line, but can not tell us whether we are to the left or to the right of the line, so we do not know which way we should be turning.

We could program the robot, once it gets off the line, to start turning first left, then right, until it sees the line. But it is not easy to program and the result is rather inefficient — so we need a better solution.

Here is one idea: let us have the robot, when it is on the line, steer slightly to the right instead of going straight. This way, we will, of course, get off the line — but then we will know for certain on which side of the line we are (right), so it should be easy to get back — by steering to the left. This gives the following algorithm:

```
task main(){
  while (true) {
    go forward steering to the right until we are off the line
    go forward steering to the left until we are back on the line
  }
```

This is not, of course, a RobotC program yet — it contains parts which need to be translated from human language to RobotC. However, before we do that, let us see whether the program can be improved. The way it is written, each statement " go forward.. until..." would require a `while` loop, so we will have three `while` loops in the program. Can we make is simpler?

Analyzing what the program does, we see that while the robot is on white (i.e., on the line), it steers to the right, and while it is on black, it steers to the left. Thus, the program can be simplified:

```
task main(){
  while (true) {
    if (robot is on white) {
      // robot is on the line
      go forward steering to the right
    } else {
      // robot is off the line -- must be to the right of the line
      go forward steering to the left
  }
}
```

We now need to translate it to RobotC. This is easy: to have the robot steer to the right, we need left motor to have more power than the right. This gives the following program

```
#pragma config(Sensor, S1, lightSensor, sensorLightActive)
task main(){
  while (true) {
    if (SensorValue[lightSensor]>40) { // robot is on the line
      motor[motorB]=60; //go forward steering to the right
      motor[motorC]=20;
    } else { // robot is off the line -- must be to the right
      motor[motorB]=20; //go forward steering to the left
      motor[motorC]=60;
  }
}
```

(as before, if using a color sensor instead of the light sensor, replace `sensorLightActive` by `sensorCOLORRED`).

The cutoff value of 40 for the light sensor was found experimentally, by checking the values returned by the light sensor on black and white surfaces (see Section 6.4). To choose the values for motor power, we need to take into account several things:

- The robot should not go too fast — otherwise it will not have time to react to the readings of the light sensor. Thus, we have chosen the average motor power to be 40.
- The difference in motor power for left and right motor dictates how much the robot is turning. If the difference is small, the robot will go almost straight — which means that if the line makes a sharp right turn, the robot may miss it and end on the left side of the line, which will cause the program to fail. If the difference is too large, the robot will be mostly turning left and right making very little headway. so the program will be inefficient. The values of 20 and 60 seem like a reasonable compromise — but you can experiment with other values.

You can download this program and test it. If it doesn't work as expected, try changing the numbers 20 and 60. Also, the program works best if the light sensor is some distance (at least 2 inches) to the front of the wheels — you may move it forward by attaching an extra beam.

Chapter 4

Variables and Constants

Variables are one of the key concepts of programming. As in mathematics, a variable is a named quantity which can take different values. One can think of the variable as a box which has a name (say, `x` or `TotalDistance`); inside this box we can put any number (or possibly some other type of data).

4.1. Variable Declaration

To use a variable in the program, we must *declare* it, telling the program the name of the variable and what type of values it will take (integers, symbols,...). The syntax of variable declaration is

```
type variable_name;
```

A variable must be declared before it is used; thus, it is common to declare a variable in the beginning of the program, right after `task main`. Here is an example of a variable declaration:

```
task main(){
  int n;
  ...
}
```

This declares a variable `n` of type `integer`.

Commonly used types are listed in Table 1. We will only be interested in integer, float, and boolean types. Note that physical limitations of NXT brick impose some restrictions on the range of values. For example, integer value must be between -32,768 and 32,767.

Type	Full name	Description	Examples
`int`	integer	positive and negative whole numbers (and zero)	`3`, `0`, or `-1`
`float`	float	real number in decimal form	`3.14`, `2`, or `-0.33`
`bool`	boolean	a value that is either true or false	`true`, `false`
`char`	character	a single character	`v`, `H`, or `2`
`string`	string	a sequence of characters (possibly empty)	`Georgia`, `house`, or `a`

Table 1. The five basic datatypes.

Variable names must satisfy some rules:

- Only letters, digits, and underscores `_` are allowed. In particular, a variable name can not have spaces. Note that variable names are case-sensitive.
- The first symbol must be a letter
- It must be different from existing variables, commands, and special words of the language such as `task`.

It is recommended that you give variables meaningful names, even if it makes them longer. It may be OK to have a variable named `m` or `n` if it is only used temporarily, but for a variable which is used many times in the program, a name like `TotalDistance` is a lot more helpful than `d`.

You can combine several declarations in a single statement:

```
int m1, m2, n;
float x, y;
```

4.2. Assignment

To give a value to a variable, you use assignment operator, `=`:

```
variable name=some expression;
```

for example, `x=5.1;`. The right-hand side can be any expression which gives the value of the same type as the variable, so one can write `x=(y+z)/2;` if `x`, `y`, `z` are floats. More details on what operations can be used in expressions are given in Section 4.4.

Note that the right-hand side can contain the same variable you are assigning the value to, for example:

```
x=x+1;
```

This looks strange to a mathematician, but makes perfect sense in programming: remember that `=` is not equality but assignment operator. What this command would do is to take the current value of `x`, add one to it, and then assign the resulting value to `x`. In other words, it would increase the value of `x` by 1. (By the way, this operation is so common that there is a special shortcut for it: command `x++;` is the same as `x=x+1;`)

You must always assign some value to a variable before you use it. If you write something like this:

```
int n;
motor[motorB]=n;
```

the results are unpredictable. To simplify things, RobotC allows one to assign initial value to a variable in the variable declaration:

```
int n=0, k=1;
```

4.3. Named Constants

In addition to variables, RobotC also allows you to introduce named constants, denoting, for example, the number 3.1415926 by `PI`. Constants are similar to variables, the only difference is that constants can not be changed — you can not assign a new value to the constant (if you try, you get an error message).

To introduce a new constant, it must be declared using the following syntax:

```
const type CONSTANT_NAME = value;
```

For example,

```
const float FIELD_WIDTH = 210.0;
```

The naming rules are the same as for ordinary variables. Traditionally, constants are named using upper case letters and are declared in the very beginning of the program, before `task main ()`.

A common use of named constants is for parameters describing the physical properties of the robot or the field. For example, in the line follower program we checked whether the robot is on the black line by using condition `SensorValue[lightSensor]<40` . The cutoff value of 40 was found experimentally; it depends on the placement of the sensor, field material, ambient light, etc; if we move the robot to a different field, we might need to change the value. In a more complicated program, if the condition is used in many places, we would have to find all occurrences of the number 40 and replace it.

A better way is to introduce a named constant `LIGHT_THRESHOLD` using the the declaration like this

```
const int LIGHT_THRESHOLD = 40;
```

and whenever we need to check that the robot is on black, use condition `SensorValue[lightSensor]<LIGHT_THRESHOLD`. In this case, if we need to change the threshold value from 40 to say 42, we would only need to change one line in our program, the constant declaration. Similarly, the difference of powers of the two motors in the line follower program (which determines how much the robot is steering to one side) can also be made a named parameter:

```
#pragma config(Sensor, S1, lightSensor, sensorLightActive)
const int LIGHT_THRESHOLD=40; //cutoff value between white and black
const int STEERING=20; //increase this to make robot turn more
task main(){
  while (true) {
    if (SensorValue[lightSensor]>LIGHT_THRESHOLD) {
      // robot is on the line
      motor[motorB]=40+STEERING; //go forward steering to the right
      motor[motorC]=40-STEERING;
    } else {
      // robot is off the line -- must be to the right of the line
      motor[motorB]=40-STEERING; //go forward steering to the left
      motor[motorC]=40+STEERING;
  }
}
```

RobotC comes with one named constant declared, `PI=3.1415926`.

4.4. Operations

You can create complicated expressions using variables, constants, operations such as `+`, and parentheses. Here are some of the more common operations and functions:

Arithmetic operations. You can use the usual arithmetic operations (+, -, /, *) and parentheses. They work in RobotC in the usual way with two exceptions:

- Multiplication is denoted by `*` and you can not omit the multiplication sign: unlike usual algebra, you can not write `2x` instead of `2*x`
- When working with integers, division works in a special way: it automatically detects when it is operating on a pair of integers and, in that case, switches to whole number division. **Whole number division** returns an integer value that represents the number of times the denominator goes into the numerator, dropping any remainder. For example, the expression `3/2` in RobotC evaluates to `1`, not `1.5`. The division operator only performs whole number division if *both* the numerator and denominator are integer type variables or literals. In all other cases, ordinary division is used. To force ordinary division of two integers either include a decimal point if it is a literal value, e.g. `3/2.0` instead of just `3/2`, or convert the integer variable to a float by preceding it with `(float)`, e.g. `((float)n)/2` instead of `n/2`.

If we want the remainder after whole number division, there is separate operator for integers, `%` (modulus) that returns the remainder upon division of the left-hand side by the right-hand side. For example, `53%10` evaluates to 3.

If this is not enough, you can also use a wider range of mathematical functions, from square roots to trigonometric functions to logarithms — check RobotC documentation under NXT FUNCTIONS→MATH. RobotC also provides a random number generator.

Comparison operations. The comparison operators take one or more inputs (usually numbers, but they work with other data types as well) and produce a boolean (true/false) value. They include:

Operation	Meaning
`a<b`	a is less than b
`a>b`	a is greater than b
`a<=b`	a is less or equal than b
`a>=b`	a is greater or equal than b
`a==b`	a is equal to b
`a!=b`	a is not equal to b

Warning. Note that when comparing whether two values are equal, you use **two** equality signs, `==`, to set it apart from the assignment operator. Thus, writing

```
if (x=5) {
  ...
}
```

is incorrect: instead of comparing `x` with 5, the expression in parentheses would **assign** to `x` value of 5 (and return true, for technical reasons).

Logical operations. For these operations, both the inputs and the result of operation are boolean (true/false)

Operation	Meaning
`P && Q`	P and Q
`P || Q`	P or Q
`!P`	not P

Note that logical `or` in the table is the so-called "inclusive or": it is true if at least one of `P`, `Q` is true (in particular, it is true if both `P` and `Q` are true).

Here is an example: a `while` statement using some of the operations above:

```
while ( ( x+y>10)|| (x>7) || (y>7) ) {
...
}
```

This loop will be performed while at least one of three conditions in parentheses is true.

Order of operations. As in usual algebra, RobotC has rules for determining the order of operations. For arithmetic operations, the rules are the same as in the rest of mathematics. Order of operations for logical operations and comparisons can be found in RobotC documentation. If you are in doubt, use parentheses to enforce the desired order of operations. For example, it is not immediately clear in which order will the operations be performed in the expression `x==5 && y>2 || y<0` , so it would better to write `(x==5) && ( (y>2) || (y<0))`.

4.5. Arrays

In some situations you need to use many similar variables (for example, if you make many measurements of light intensity and want to keep them all). You can, of course, introduce several variables with names like `x0`, `x1`, `x2`..., but there is a better solution: you can use arrays. An array is a kind of a varible which, instead of storing a single value, can store a whole sequence of values; if a usual variable can be thought of as a box, an array is a box with many compartments indexed by numbers. We refer the reader to RobotC documentation (or to any decent programming book) for discussion of how one declares and uses arrays.

Chapter 5

Functions

5.1. Creating New Functions

When writing a program, it is common that the same block of code is used in many places. For example, in writing the program for the robot we repeatedly use the sequence of commands `motor[motorB]=100; motor[motorC]=100;` to start move forward. Similarly, the group of commands that tells the robot to turn 90deg left or right is also used a lot in many programs.

Instead of copying and pasting the text, RobotC (as well as any other computer language) allows one to "encapsulate" such a group of commands, essentially defining a new command and giving it a new name, so that any time we need that block of code, we can just use that name instead of cutting and pasting the whole block. The proper name for this construction is "function". Here is the simplest example of a program using a function.

```
// Start motors forward
void StartForward(){
  motor[motorB]=100;
  motor[motorC]=100;
}
//turn 90 degrees right
void TurnRight() {
  motor[motorB]=100;
  motor[motorC]=-100;
  wait1Msec(400); // this value was found experimentally
  motor[motorB]=0;
  motor[motorC]=0;
}
task main() {
  while (true) {
    StartForward();
    wait1Msec(2000); // go forward for 2 seconds
    TurnRight();
  }
}
```

Here is the explanation of the program. The parts `void TurnRight() { ... }`, `void StartForward(){ ... }` are *declarations* of new functions: we define new functions with the names `TurnRight` and `StartForward`. Everything between the curly braces in the function declaration is the body of the function: every time we use (or, in programmers jargon, call) the function, these commands are executed. Note that the function declaration must be given before the function is used, so normally it is done in the beginning of the program, before `task main()`.

In the body of the program we just use these function names in the same way as we would use any other command — except that we need to include the parentheses: `TurnRight();`. The importance of this will be explained later.

Note that in this program we are only using each command once, so we are not really saving any typing by using functions. But even in this case, using functions offers several advantages. For one thing, the program is now much easier to read and understand. For another, if we later find that turning 90 deg requires running the motors not for 400 ms but say for 419 ms, we only need to change the time in the declaration of `TurnRight` function — the rest of the program need not be changed.

Note. *Note that doing turns by turning motors on for a specified time is not very precise. We will discuss more reliable ways of controlling the robot movements and turns in Chapter 7.*

5.2. Functions With Arguments

We frequently need to have a function whose action depends on some "input values", or arguments. For example, it is convenient to have a function that would make the robot move forward for a given time (measured in milliseconds). It can be easily done in RobotC with the following function declaration:

```
// Go  forward for t milliseconds
void MoveForward(int t){
  motor[motorB]=100;
  motor[motorC]=100;
  wait1Msec(t);
  motor[motorB]=0;
  motor[motorC]=0;
}
```

The words `int t` in parentheses after function name show that the function expects one argument, or one input value, which must be of type `integer`, and that inside the function body we will use the variable `t` to denote this argument.

To use this function, you should provide the argument: for example, `MoveForward(2000);` will move your robot forward for 2 seconds. You can also use as an argument any expression that gives integer value: e.g., if `a` and `b` are integer variables, you can use `MoveForward(a+b)`. If you forget to provide an argument, typing `MoveForward()`, you will get an error message.

You can also have functions that use more than one argument. For example, the following function would move at given power for given time:

```
// Go straight at given power  for t milliseconds
void Move(int power, int t){
  motor[motorB]=power;
  motor[motorC]=power;
```

```
  wait1Msec(t);
  motor[motorB]=0;
  motor[motorC]=0;
}
```

It can be used like this: `Move(75, 3000);`. Note that the order of arguments is important: if you typed `Move(3000, 75);`, you would get something quite different.

Now we can explain the meaning of pair of parentheses used to call functions without arguments: according to the rules of RobotC, every function has arguments — but the list of arguments can be empty, so we have parentheses with nothing inside.

5.3. Return Values

Functions in RobotC can have one more feature: they return a value. In fact, this is the usual meaning of the word function in mathematics: a function (such as $\sin(x)$) is something that takes as input one or more arguments and produces, or returns, a single value.

To have a function that returns a value, you need to describe what type of value it returns in the function declaration, and include in the body of the function the command `return(value)`. Here is the simplest example of such a function. This function reads the value of the light sensor, `lightSensor` (assuming that it has been configured using appropriate `#pragma...` statement) and returns either `true` or `false` depending on whether the value is larger than the cutoff value.

```
bool RobotOnWhite(){
  if (SensorValue[lightSensor]>40) {
    return(true);
  } else {
    return(false);
  }
}
```

The word `bool` means that the value returned by the function is of type `boolean`.

This function could be used as follows:

```
while ( RobotOnWhite() ) {
  ...
}
```

Again, the benefits of this are a) increased readability of the code and b) if we need to change the cutoff value (say, from 40 to 35), we only need to do this at one place.

Now we can explain the meaning of the word `void` in declarations of functions in the previous sections: it is there to indicate that the function provides no return value.

Note that a function can only return one value. If for some reason you might need a function that returns more than one value, you can achieve it in several ways. The easiest way is to have the function modify the values of global variables as described in the next section.

5.4. Local and Global Variables

Inside a function, you can declare and use new variables. However, these variables will only be available inside the function; you can not access their values elsewhere in the program. Consider the following example:

```
//Compute the mean of two numbers
float mean(float a,b) {
  float x;
  x=(a+b)/2;
  return(x);
}
task main(){
  float y;
  y=mean(2.5, 3.5) + 1;
  ...
}
```

In this example, if you tried using variable `x` inside `task main()`, you would get an error. Similarly, variable `y` was declared inside `task main` and is only available inside `task main` — in particular, it can not be used in the body of function `mean`, since the declaration of this function is outside of `task main`. One says that variable `x` is a local variable of the function `mean`, and `y` is a local variable of `task main`.

There are very good reasons for this separation. It is common that a function written by one programmer is used by many others in different programs, so a person writing the function can not know where it will be used and what variables will be declared in the programs that use this function. Similarly, people using the function need not know its inner working and what variables it introduces — they only need to know what inputs it expects, what actions it performs and what value it returns, not how it achieves this. Making local variables unavailable outside the function or task where they are defined enforces this separation.

Another good reason for the separation is that this allows one to avoid naming conflicts. We can have several functions, each using the same variable name (say `x`) for some temporary storage, without fear that these functions will interfere with each other. Consider this example:

```
float f() {
  float x;
  ...
}
float g() {
  float x;
  ...
}
```

In this case, both functions `f` and `g` use variable `x`. However, they will not interfere with each other, even if they are called simultaneously (yes, this is also possible with RobotC — see Chapter 8). Each function has its own variable `x`; when `x` is used inside body of `f`, it refers to "`x` of function `f`"; when `x` is used inside body of `g`, it refers to "`x` of function `g`". These two variables are not related in any way; they can have different values.

One can also declare a global variable, which would be available everywhere in the program, by declaring them at the very beginning of the program, outside any function or task:

```
float z;// z is a global variable, available anywhere in the program
//Compute the mean of two numbers
float mean(float a,b) {
  float x;
  x=(a+b)/2;
  // one can use z here
  return(x);
}
task main(){
  float y;
  y=mean(2.5, 3.5) + 1;
  // one can also use z here

  ...
}
```

It is advised that you only declare a variable as a global variable if you really need it; normally, variables should be declared inside the function or task that uses it.

Chapter 6

Sensors

6.1. NXT Sensors

In addition to the light sensor discussed before, NXT set also contains other sensors. Here is a full list of sensors included with NXT set:

Touch sensor: This sensor has a button that can be pressed. It is commonly used to detect when the robot bumps into an obstacle such as a wall. Usually one builds some kind of a bumper which would press on the touch sensor when the robot hits the wall.

Sound sensor: This sensor (only included in NXT 1.0 set) measures the sound intensity, so one can make the robot react to loud noises such as claps. Note that it can only measure the intensity of the sound; you can not use it to recognize spoken commands. In my experience, it is the least useful of all the sensors.

Light sensor: This sensor is included in NXT 1.0 set. It measures the light intensity and can be used in two modes:

- Inactive mode (sometimes also called *ambient light mode*), when the sensor just measures the intensity of light falling on it.
- Active mode (also called the *reflected light mode*), when the sensor also acts as a light source, using a built-in red LED light. It is very useful when the sensor is close to some surface (say, the floor), as in this case it can determine how dark the surface is by measuring the reflected light.

Color sensor: This sensor was introduced in NXT 2.0 set, replacing the light sensor. It has three built-in LEDs, emitting red, green, and blue light. Comparing the readings of reflected light for each of colors, one can determine the color of an object.

Ultrasonic (sonar) sensor: This sensor can be used to determine the distance to the nearest solid object. It does so by sending out pulses of ultrasonic sound (i.e., sound with frequency too high for the human ear to hear), listening for echoes and measuring the time interval between the original pulse and the echo. It is reliable at ranges from 7–150 cm for large objects (such as walls), even though the precision is not very high (officially, it is $\pm$3cm; your results might be different). For smaller objects (less than 5 cm in diameter), it is less useful.

Note that if you need to use several ultrasonic sensors at the same time, their signals might interfere with each other, so you might not get a reliable reading. To overcome this

problem, you need to use the ultrasonic sensors in "Single Shot Mode", which is only possible with the use of third party sensor drivers. An example can be found in Ray McNamara's blog: `http://www.rjmcnamara.com/lego-minstorms/robotc-multiple-utrasonic-sensors/`

In addition to these sensors, one can also use a variety of sensors by other manufacturers, most notably by HiTechnic (`http://www.hitechnic.com`). These include infrared sensor, gyro sensor, and much more. To use third party sensors in RobotC, you will need a separate driver pack available here: `http://botbench.com/blog/robotc-driver-suite/`.

6.2. Setting up And Using Sensors in RobotC

To use a sensor in RobotC, you first need to let the program know what sensor you are using. The easiest way of doing this is by using Motor and sensor setup tool. To launch it, select Robot→Motor and Sensor Setup, then click on `Sensors` tab. You will get a window like the one shown in Figure 6.1

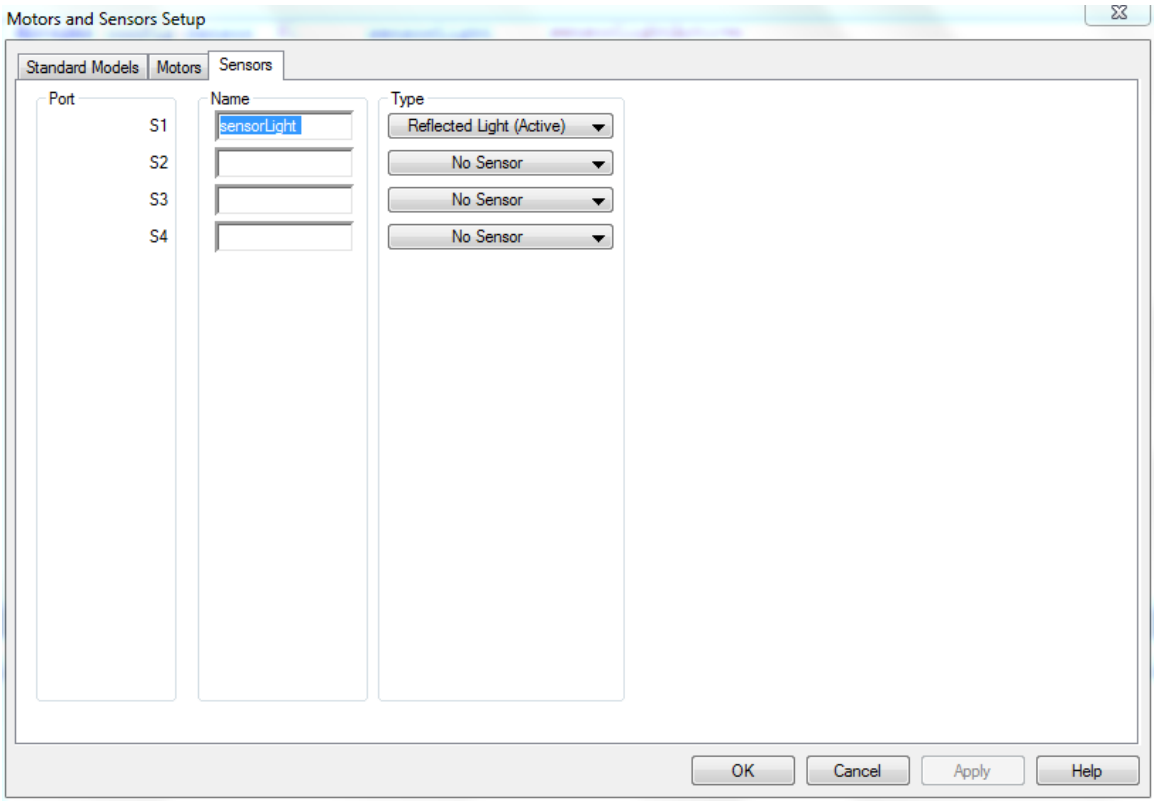

Figure 6.1. Sensor Setup window

In this window, you can select what sensors you have connected to different ports of NXT (S1 stands for "port 1", etc) and give each sensor a name, such as `TouchSensorLeft`. After selecting the values, click `OK` and the tool will write the configuration to your program, creating lines like

```
#pragma config(Sensor, S1, lightSensor, sensorLightActive)
```

You can also create or modify these configuration lines manually, without using the setup tool, but it is only recommended for experienced users.

After configuring the sensors, you can use them in your program. Namely, the expression `SensorValue[SensorName]` (where `SensorName` must be replaced by the name you gave to the corresponding sensor in the setup tool) returns the current reading of the sensor. The meanings of these readings are as follows:

- For touch sensor: possible values are 0 (not pressed) and 1 (pressed). Note that these values can be used directly in conditional statements: `if (SensorValue[sensorTouch]) {... }`. RobotC will convert 0 (not pressed) to `false` and 1 (pressed) to `true`.
- For sound sensor: the value is the sound intensity in decibels over the standard threshold of hearing.
- For light sensor: the value measures the light intensity on the scale 0-100.
- For color sensor: see next section.
- For ultrasonic (sonar) sensor: the value ranges 0–255 and shows the distance to the nearest solid object in cm. If no object was found (i.e., no reflected sound waves detected), the value will be 255.

6.3. Color Sensor

Using a color sensor is more complicated. This sensor contains a single light sensitive element and three LEDs, emitting red, green, and blue lights respectively. It can be used in one of 5 modes described below. For each mode, the name in parentheses shows the notation for this mode used in `#pragma config(... )` statement.

Ambient (`sensorCOLORNONE`): In this mode, all LEDs are off, and the sensor measures the ambient light (same as the light sensor in inactive mode). The value returned by `SensorValue[colorSensor]` shows light intensity on the scale 0-100.

Red (`sensorCOLORRED`), Blue (`sensorCOLORBLUE`), Green (`sensorCOLORGREEN`): In each of these three modes, only one LED is active (of the corresponding color), and the sensor measures the reflected light. The value returned by `SensorValue[colorSensor]` shows light intensity on the scale 0-100. These modes are useful for detecting light/dark areas on the surface. In particular, in `sensorCOLORRED` mode the sensor is quite similar to the light sensor in active mode.

RGB (`sensorCOLORFULL`): In this mode, the sensor uses (in turn) all of the LEDs and measures the reflected light for each of them; it then uses this information to determine the color of the object. The value returned by `SensorValue[colorSensor]` can take one of six values: `BLACKCOLOR`, `BLUECOLOR`, `GREENCOLOR`, `YELLOWCOLOR`, `REDCOLOR`, `WHITECOLOR`. Note that these are not strings but named constants.

The mode can be changed in the program by using the command `SensorType[sensorname] = mode`, where `sensorname` is the name you gave to the sensor in the setup tool and `mode` is the machine name of the mode, for example:

```
SensorType[colorSensor] = sensorCOLORFULL;
```

Here is an example of a simple program that makes the robot go forward until it detects a red or blue line:

```
#pragma config(Sensor, S1, colorSensor, sensorCOLORFULL)
task main () {
  motor[motorB]=100;
```

```
  motor[motorC]=100;
  while ( (SensorValue[colorSensor]!=REDCOLOR) &&
          (SensorValue[colorSensor]!=BLUECOLOR)
        ) {
    //do nothing -- just wait for red or blue color
  }
  motor[motorB]=0;
  motor[motorC]=0;
}
```

The expression `(SensorValue[colorSensor]!=REDCOLOR)&&(SensorValue[colorSensor]!=REDCOLOR)` is true while the color detected by the robot is neither red nor blue. Thus, the while loop would stop once the color detected is either red or blue.

For experts, we mention that when the color sensor is in `sensorCOLORFULL` mode, it is possible to get information about reflected light intensity for each color separately. To do it, we need more than just `SensorValue` function. The program below shows how it can be done; read the comments to see what is going on. It uses arrays (see Section 4.5) and command `nxtDisplayTextLine` (see Section 9.2) to print information on NXT screen.

```
#pragma config(Sensor, S1, colorSensor, sensorCOLORFULL)
task main () {
  short nColorValues[4]; //an array of 4 short integers (each 0-255)
  int R, G, B, Amb;
  while (true) {
   // get all color values at once
    getColorSensorData(colorSensor, colorValue, &nColorValues);
    //get and display individual values:
    // Reflected - red LED
    R=nColorValues[0];
    nxtDisplayTextLine(3, "%s:␣%4d", 'Red', R);
    // Reflected - green LED
    G=nColorValues[1];
    nxtDisplayTextLine(4, "%s:␣%4d", 'Green', G);
    // Reflected - blue LED
    B=nColorValues[2];
    nxtDisplayTextLine(5, "%s:␣%4d", 'Blue', B);
    // ambient light level - all LEDs off
    Amb=nColorValues[3];
    nxtDisplayTextLine(6, "%s:␣%4d", 'Ambient', Amb);
  }// end of while loop
}
```

6.4. Testing the Sensors

RobotC allows you to test the attached sensors, showing the current reading for each of them. This is very useful when determining the cutoff values or testing the sensor placement on the robot.

There are two ways of testing the sensors: using NXT's built-in view function and using RobotC.

To view current sensor value using NXT built-in functions, select in the main NXT menu VIEW, then select the appropriate sensor and port. Using this method, you can only view the readings of one sensor; on the plus side, the robot does not have to be connected to the computer.

To view the current sensor values using RobotC, make sure the NXT brick is on, connect it to the computer and then select in RobotC menu ROBOT→NXT BRICK→POLL BRICK. It will produce a window similar to the one in Figure 6.2. Click on `More` button to configure the tool, by selecting

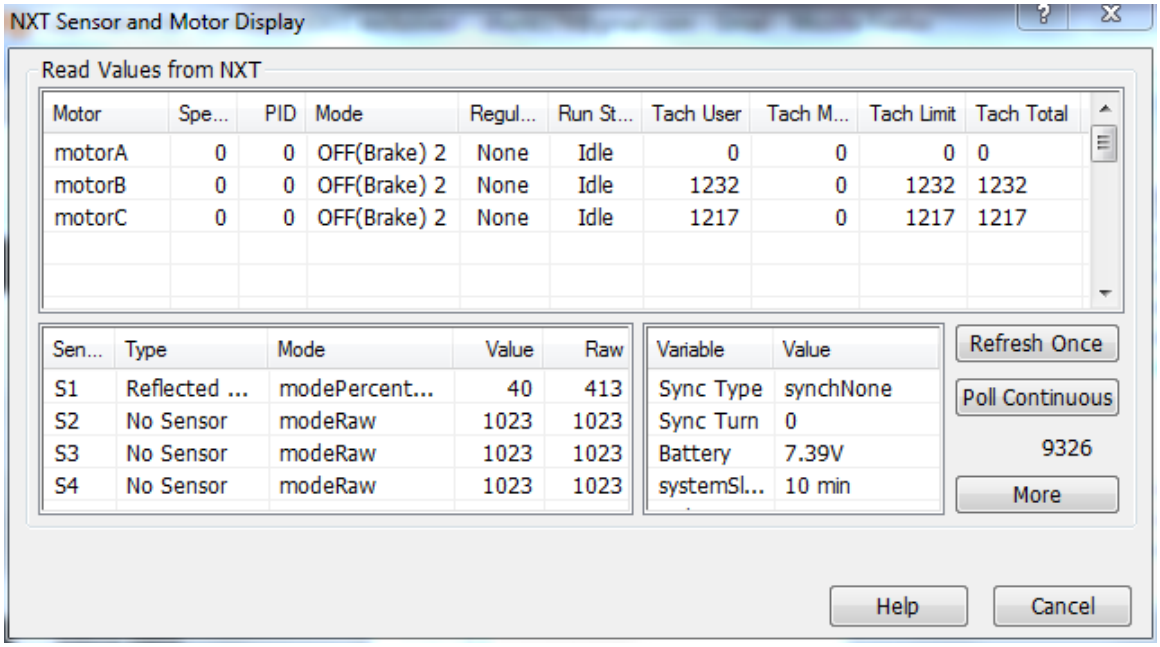

Figure 6.2. Poll brick tool

the types of sensors attached to each port. You will also need to select a "mode" for converting raw values (which is what the sensor actually sends back to NXT brick; these values range from 0–1023) to numerical values like the ones you would get using `SensorValue[..]` function. For touch sensor, choose `boolean`; for light sensor, choose `percent`; for sonar sensor, choose `raw`.

After configuring the sensors, click on `Refresh Once` to read values once or on `Poll Continuous` to get continuously updated values. The current value for each sensor will be shown in `Value` column (not in the `Raw` column).

Of course, the robot must be connected to the computer by a USB cord (or via Bluetooth).

Chapter 7

Motors and Movement

We have already seen the simplest commands for controlling the motors: setting motor power with commands

```
motor[motorB]=power;
```

where a `power` can be any number between -100 (representing going full speed backwards) and 100 (going full speed forward).

The drawback of these commands is that the actual rotation speed depends not only on the power given to the motor but also on the battery charge level, motor load, individual motor characteristics and more. Thus, these commands are not suitable if you want the robot to move some specified distance.

To overcome this, LEGO motors have built-in rotation counters (called encoders). This can be used to make the robot move for specified distance — which is a lot more useful than specifying time. They can also be used to have two motors rotate in sync, at exactly the same rate — which is commonly used if the motors control left and right wheels of the robot.

The low-level commands for working with motor encoders are discussed in Section 9.6. Most of the time, however, one needs higher level commands such as "move forward for 25 cm". Such a set of commands is sometimes referred to as the "natural language". RobotC includes one such set of commands (it is not enabled by default); here we discuss a different set of commands, created by the author of this primer. Note that this set is an alternative to RobotC's "natural language", so when using it, there is no need to enable RobotC's natural language.

7.1. Natural Language: Setup

For this primer, we have created a set of functions for controlling the motors and movement. To use these functions in your program, you need to do the following:

(1) Download the file `nat_language.c`, containing these functions from `http://islandbots.org/robotc/` and place it in the same directory as your program.

(2) Include in your program the following lines

```
float MOVEUNIT=21.1;
float TURNUNIT=2.12;
#include "nat_language.c"
```

These lines should be placed in the very beginning of the program, outside of `task main` and before any function declarations. The `#include` line instructs the program to include the function definitions from the file `nat_language.c` (we will discuss include statements later, in Section 9.4). Note that the file name is enclosed in double quotes. The other two lines define values of some variables used by these functions. Later you might want to modify the values, see Section 7.3.

(3) In the beginning of `task main`, include the line

```
NLSetMotors(leftMotor, rightMotor);
```

where `leftMotor`, `rightMotor` are the motors controlling left and right wheel/track of your robot, for example: `NLSetMotors(motorB, motorC);`

If you find that command like `NLStartForward` (described later) actually make your robot move backward, add the word `REVERSED` as the third argument to the `SetMotors` command:

```
NLSetMotors(motorB, motorC, REVERSED);
```

(note: the word REVERSED is a name of some constant defined in the file `nat_language.c`, so it should be entered exactly as written, without quotes)

You are now ready to use the functions of natural language.

7.2. Natural Language Functions

Here is the list of functions included in the natural language. All of them begin with letters NL, to avoid possible confusion with similar-looking commands of RobotC.

Motion functions. Table 1 below lists the functions used for controlling the motion

The parameter `power` in all the commands sets the power of the motors. It must be an integer between 0-100 (do not use negative values). It is optional: it can be omitted, in which case the default value of 75 is used. For example, writing `NLStartForward()` is the same as `NLStartForward(75)`.

For moving forward/backward, the distance is specified in certain units. The length of the unit depends on the value of variable `MOVEUNIT`. The default value (21.1) is chosen so that when using the standard LEGO wheels of NXT 1.0 set, one unit is equal to 1 cm. If you are using different wheels, you may need to change the value of `MOVEUNIT` to make it equal to 1cm; see Section 7.3.

The command `NLRotatMotor` rotates the motor by the given angle. It can create problems: if something prevents the motor from rotating that far (for example, it hits some part of the robot), the program will continue trying to turn the motor and will not move to the next command until the rotation completes (or until something breaks).

To avoid getting stuck, in such situations you can use the command `NLSafeRotatMotor`, which rotates the motor by given number of degrees or given time, whatever comes first. For example, command `NLSafeRotateMotor(motorA,90,0.7, 50)` will try to rotate motor A by 90 degrees at 50% power, but if it can not complete the rotation, it will stop after 0.7 seconds. (Note that under normal conditions, 0.7 seconds is more than enough time to rotate motor by 90 degrees).

Turns. The table below lists the functions used for turning the robot.

Functions `NLTurnLeft`, `NLTurnRight` make the robot turn in place, by making one wheel go forward and the other backward. Thus, the center of rotation will be the midpoint between the wheels. Function `NLPivotTurnLeft` turns the robot left by having the left wheel stopped and

Function	Description	Arguments
`NLRotateMotor(motor, deg, power)`	Rotates one motor given number of degrees	`motor`: motor to be rotated (e.g. `motorA`) `deg`: number of degrees (can be negative) `power` (optional)
`NLSafeRotateMotor(motor, deg, time, power)`	Rotates one motor given number of degrees or time, whichever comes first	`motor`: motor to be rotated (e.g. `motorA`) `deg`: number of degrees (can be negative) `time`: time (in seconds) `power` (optional)
`NLStartForward(power)` `NLStartBackward(power)`	Starts both drive motors going forward/backward	power (optional)
`NLStopMotors()`	Stops both drive motors	
`NLGoForward(dist, power)` `NLGoBackward(dist, power)`	Move forward/backward for specified distance, using drive motors	`dist`: distance (in MOVEUNITS; 1 MOVEUNIT $\approx$ 1 cm) `power` (optional)
`NLGoForwardTime(time, power)` `NLGoBackwardTime(time, power)`	Move forward/backward for specified time, using drive motors	`time`: time (in seconds) `power` (optional)

Table 1. Natural Language movement functions

Function	Description	Arguments
`NLTurnLeft(units, power)` `NLTurnRight(units, power)`	Turn left/right in place, rotating wheels in opposite directions	`units`: number of units to turn. 1 unit $\approx$ 1 degree `power` (optional)
`NLPivotTurnLeft(units, power)` `NLPivotTurnRight(units, power)`	Do a pivot left/right turn	`units`: number of units to turn. 1 unit $\approx$ 1 degree `power` (optional)

Table 2. Natural Language turns

going forward with the right wheel; thus, the center of rotation is the left wheel. Similarly, for `NLPivotTurnRight`, the robot turns around the right wheel.

For both kinds of turns, the magnitude of rotation is measured in units; how large one unit is is determined by the variable `TURNUNIT`. The default value (2.12) is chosen so that for a typical robot (NXT 1.0 wheels, set 12 cm apart), one unit=1 degree of robot rotation. Again, you might want to change the value — see Section 7.3.

As before, parameter `power` in all the commands sets the power of the motors. It must be an integer between 0-100 (do not use negative values). It is optional; if it is omitted, the default value of 75 is used.

Wait functions. The table below lists the functions used to wait for a specified event. Some of these functions use sensors; before using these functions, you must configure the corresponding sensor as described in Chapter 6.

Function	Description	Arguments
`NLWaitForButton();`	Wait until the orange button on the NXT is pressed	
`NLWaitForTouch(sensor);`	Wait until the touch sensor is pressed	`sensor`: touch sensor name
`NLWaitForLight(sensor,cutoff);`	Wait until the reading of the light or color sensor gets above the specified cutoff value	`sensor`: light/color sensor name `cutoff`: the cutoff value (0-100)
`NLWaitForDark(sensor,cutoff);`	Wait until the reading of the light or color sensor gets below the specified cutoff value	`sensor`: light/color sensor name `cutoff`: the cutoff value (0-100)

Table 3. Natural Language wait functions

7.3. Calibrating the Moves/Turns

As was mentioned before, the default values of `MOVEUNIT` and `TURNUNIT` are chosen so that for a typical 2-wheel robot, 1 `MOVEUNIT` $\approx$ 1 cm, and 1 `TURNUNIT` $\approx$ 1 degree of turning. However, it depends on the robot construction, so it may happen that for your robot, these values are off, and you need to adjust them.

To adjust the value of the `MOVEUNIT`, write a short program that makes the robot execute the command `NLGoForward(100)`. Measure the distance the robot has travelled. If the distance is less than 100 cm, the value of `MOVEUNIT` is too low and needs to be increased; if the distance is larger than 100 cm, the value is too high and needs to be decreased. For example, if the distance traveled is 95 cm, it means that the value of `MOVEUNIT` needs to be increased by the factor of $\frac{100}{95} \approx 1.05$

Similarly, to adjust the value of the `TURNUNIT`, write a short program that makes the robot execute the command `NLTurnLeft(360)`. If the robot turned less than one full rotation, the value of `TURNUNIT` is too low and needs to be increased; if the robot turned more than one full rotation, the value of `TURNUNIT` is too high and needs to be decreased.

7.4. Sample Program

Here is a sample program using the natural language

```
#pragma config(Sensor, S1, lightSensor, sensorLightActive)
float MOVEUNIT=21.1;
float TURNUNIT=2.12;
#include "nat_language.c"
const int LIGHT_THRESHOLD=40; //cutoff value between white and black
task main(){
 NLSetMotors(motorB, motorC);
 NLStartForward(50); // go forward at 50% power
 // wait until light sensor sees white line
 NLWaitForLight(lightSensor, LIGHT_THRESHOLD);
 NLStopMotors();
 NLTurnLeft(90);
 NLGoForward(25); // go forward for 25 cm at default power=75%
```

```
}
```

Chapter 8

Multitasking

Unlike most traditional computer languages, RobotC supports multitasking: the robot can be doing several things at the same time. For example, in the robotic sumo competition, a robot could be searching for the opponent using the ultrasonic sensor, while at the same time watching for the field boundary, to make sure it does not leave the field.

8.1. Tasks

To achieve multitasking, RobotC allows the program contain several (up to 10) *tasks*. These tasks can be running at the same time. Each task is included in the program using this this syntax:

```
task TaskName () {
 some commands...
}
```

where `TaskName` is the name of the task (which is subject to the same restrictions as variable names).

One of the tasks must be named `main`. This task will be started when the program starts; when this task reaches the end, the program stops. Until now, this was the only task we had in our programs.

The other tasks are not started automatically: they must be started using command `StartTask(taskname)` (usually given inside `task main`). One can also stop a task using command `StopTask(taskname)`.

All running tasks have access to the same motors and sensors. In addition, the tasks can communicate with each other by using global variables (see Section 5.4): a task could assign a value to a global variable, for another task to use.

Note that having several tasks giving commands to motors can lead to problems: if one task directs the robot to move forward while the other directs the robot to turn right, the results are unpredictable. Such situations should be avoided – either through the use of global variables to indicate which task has control of the motors, or simply by having one task stop all competing tasks before sending instructions to motors.

The following is a draft of a program which would have the robot follow a black line on white surface (using the algorithm from Chapter 3) until it hits a wall. After this, we back up from the wall and then stop. It uses two tasks: one to follow the line, and the other to watch for the wall.

We assume that we have a light sensor attached to port 1 and a touch sensor connected to port 2 (with appropriate bumper pressing on it when the robot hits the wall).

```
#pragma config(Sensor, S1, lightSensor, sensorLightActive)
#pragma config(Sensor, S2, touchSensor, sensorTouch)
bool HitWall=false; // will be set true when we hit the wall

// This task makes the robot follow the line
task FollowLine(){
  commands for line following algorithm
}
/*
  This task watches for the wall; once it hits, it sets the global
  variable  HitWall to true
*/
task WatchForWall() {
  while (! SensorValue[touchSensor]) {
    // do nothing - just wait
  }
  HitWall=true;
}//end of task WatchForWall

task main() {
  StartTask(FollowLine);
  StartTask(WatchForWall);
  while (! HitWall ) {
    // we have not hit the wall yet; let us just wait
  }
  // if we reached this point, we hit the wall
  StopTask(FollowLine);
  //back up from the wall, for 0.5 sec
  motor[motorB]=-70;
  motor[motorC]=-70;
  wait1Msec(500);
}
```

This program could be made shorter: we could easily do without task `WatchForWall`, using just one auxiliary task. We chose to include two tasks for educational purposes.

Chapter 9

Further tools

In this section, we give a brief overview of several other capabilities of RobotC that have not been discussed previously.

9.1. Sounds

The NXT brick has a built in speaker to play sounds, which can be used both for fun and for user feedback (for example, one can play a sound when a program reaches a certain point).

The simplest sounds one can play on NXT are just musical tones. To play a tone, use the following command

```
PlayTone(frequency, duration);
```

where `frequency` is tone frequency (in Hertz) and duration is the duration in units of 10Msec. For example, to play the tone A (440 hz) for 0.5 seconds, one would use `PlayTone(440, 50);`. You can find frequencies of musical notes online.

Note that the program doesn't wait for the sound to stop playing: whenever it meets a `PlaySound` command, it adds the sound to be played to the queue and moves on to the next command, while the NXT is playing the sound in the background. Since the queue can only hold ten sound requests, if you are playing a long tune, you should send sounds to be played in batches, waiting for one batch to stop playing before sending the next one. It can be done using built-in boolean variable `bSoundActive`: it is true while the sounds are playing and becomes false once all sounds from the queue have been played.

The following program plays the opening measure of Mozart's *Eine Kleine Nachtmusik* given in Figure 9.1.

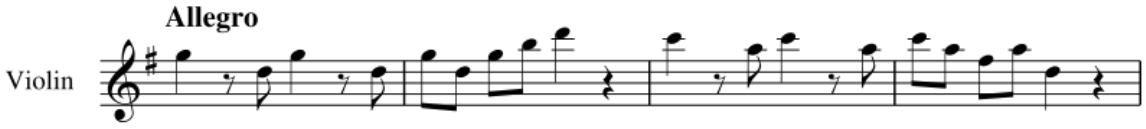

Figure 9.1. *Eine Kleine Nachtmusik*, W.A. Mozart. Using the `PlayTone()` function, the NXT can play this tune.

```
// Note Frequencies
const float D5=587.33;
```

```
const float G5=783.99;
const float F5Sharp=739.99;
const float A5=880.00;
const float B5=987.77;
const float C6=1046.50;
const float D6=1174.66;
const int BEAT=20; // in 10Msec

task main() {
 // Measure 1
 PlayTone(G5,2*BEAT);
 while(bSoundActive){} //wait for tone(s) to finish
 wait10Msec(BEAT);
 PlayTone(D5,BEAT);
 PlayTone(G5,2*BEAT);
 while(bSoundActive){} //wait for tone(s) to finish
 wait10Msec(BEAT);
 PlayTone(D5,BEAT);
 while(bSoundActive){} //wait for tone(s) to finish
}
```

Notice the use of named constants to represent the notes. Also, notice the use of the named constant `BEAT` to set the duration of notes and rests.

In addition to playing musical notes, it is possible to play other sounds. RobotC has a small set of built-in sounds (constructed of individual tones), which could be played using command `PlaySound`:

```
PlaySound(soundBeepBeep); //Play the sound 'soundBeepBeep'.
```

The list of available sounds is given below:

- `soundBlip`
- `soundBeepBeep`
- `soundDownwardTones`
- `soundUpwardTones`
- `soundLowBuzz`
- `soundFastUpwardTones`
- `soundShortBlip`
- `soundException`
- `soundLowBuzzShort`

Finally, you can also play any sound, by downloading to the NXT the sound file in a special format. However, it requires some work, and because of memory limitations, these sounds must be short (less than a few seconds), so this is not very practical. Interested readers can find the details in *Programming with Robots* by Albert Schueller, available from `http://carrot.whitman.edu/Robots/`.

9.2. Display

The NXT brick has a display that is 100 pixels wide and 64 pixels high. Unlike the latest and greatest game consoles, the display is monochrome, meaning that a particular pixel is either on or off.

It can be used both for (very simple) graphics and for text messages. To print a text message to the display, you can use the following command:

```
nxtDisplayTextLine(line_number,string);
```

where `line_number` is the line number (counted from the top; NXT allows 8 lines of text, numbered 0 through 7) and `string` is the text to be printed. In the simplest case, `string` is just a short text enclosed in quotes (you can use either single or double quote):

```
nxtDisplayTextLine(4,"Hello, World!");
```

would print the words *Hello, World!* (without quotes) on the fourth line. Note that you can only fit 16 characters on each text line.

To get larger font, you can use `nxtDisplayBigTextLine(line_number,string)`. This produces letters that are 16 pixels high (thus, they occupy two normal lines). When using this command, each line can only hold 8 characters.

For printing more complicated messages, containing computed values, one can use a different form of this command:

```
nxtDisplayTextLine(line_number,format_string, parameters);
```

where `format_string` can contain, in addition to usual symbols, also placeholders which will be replaced by the values of parameters (up to 3). Common placeholders include `%d` for an integer number, `%f` for a float, and `%s` for a string, so the following snippet of code

```
int n=5;
float x=3.14;
nxtDisplayTextLine(3,"n=%d, x=%f", n, x);
```

would print `n=5, x=3.14`.

For finer control of the output, you can also specify how many digits should be used for printing numbers:

- `%4d` will print a decimal integer, at least 4 characters wide; if the integer is less than 4 digits, blanks will be added in front: `__27`.
- `%6.2f` will print a floating point number, with at least 6 characters wide and 2 decimal places : `nxtDisplayTextLine(3,"x=%6.2f", 21.843)` will produce `x=__21.84`

Note that once the program ends, the messages it printed will disappear from the display, so make sure to include `wait1Msec(...)` to give the user time to read the message.

Warning. In RobotC 4, display commands have been renamed: instead of `nxtDisplayTextLine()`, you shoudl use `displayTextLine()`. The old version (`nxtDisplayTextLine()`) still works, but you will get a warning that it is deprecated.

9.3. NXT buttons

You can check whether any of the buttons on the NXT brick is pressed. It is useful when a program needs to wait for a user to do something before continuing; it can also be used to allow the user to select, say, one of several options. Note that due to the limitations of NXT hardware, only one button press can be recognized at a time.

To check which button is pressed, use variable `nNxtButtonPressed`. Its values are as follows:

-1 = no button pressed;

0 = Gray Rectangle button;

1 = Right Arrow button;

2 = Left Arrow button;

3 = Orange Square button.

Note that by default, pressing the gray rectangular button would stop the running program, so you can not use this button to communicate with the program; similarly, pressing arrow buttons switches between different information displays. You can override this behavior, taking over the button control; please consult RobotC help under NXT FUNCTIONS→BUTTONS.

9.4. Includes

You can have your program include another file, by entering the line `#include "FileName"`, for example

```
#include "JoystickDriver.c"
```

Note that the file name must be in double quotes, and that you do not need a semicolon at the end of the line. RobotC will look for the file with that name in two locations: the same directory as the current program and also in a system default directory (this can be overriden in preferences)

This directive will have the same effect as if you copied and pasted the contents of the file in place of the `#include` line. It is typically used when you have many functions in your program; in this case, it makes sense to move the function definitions to a separate file and `#include` it in the main program (before `task main`). This makes the main program much easier to read.

9.5. Timing

In some situations you need to know how much time has passed between two commands, or make sure that each execution of a loop takes a fixed amount of time. For example, if you want to determine how much the robot has turned, you can use a gyro sensor (it is not included in LEGO set but can be bought separately from a company called HiTechnic) which measures angular velocity, and get the total angle by adding up angular velocity multiplied by time. To do this, you need to know how mch time has elapsed between two readings of the gyro sensor.

RobotC provides 4 timers, named `T1`, `T2`, `T3`, `T4`. Each of them acts like a stopwatch: you can reset the timer to zero, or check the current value of the timer. To reset the timer, you can use command

```
resetTimer[T1];
```

To read the current value of the timer, use the expression `time1[T1]`, which returns the current value of the timer in milliseconds.

Warning. If you want to get time in seconds, do not use `time1[T1]/1000`: this would perform integer division, discarding the remainder. Thus, if the value of the timer is 2567 ms, expression `time1[T1]/1000` woudl give you 2, rather than expected float value of 2.567.

To get the float value, use `time1[T1]/1000.0` or `(float)time1[T1]/1000`.

9.6. Using encoders

As was mentioned in Chapter 7, each NXT motor has a built-in rotation counter (encoder). This is used by the natural language functions described in Section 7.2. However, if these functions are not sufficient for your purposes, you can directly use the encoders.

To reset rotation counter for a given motor to zero, you can use the command

```
nMotorEncoder[motorB]=0;
```

(obviously, you can replace `motorB` by `motorA` or `motorC`).

To get the current value of rotation counter for a motor, use the expression `nMotorEncoder[motorB]`. It measures the number of rotations (in degrees). Note that it can be either positive or negative: rotations in one direction count as positive, and in the opposite direction, as negative.

Thus, the following snippet of code would run motor B for exactly 3 rotations (=1080 degrees):

```
nMotorEncoder[motorB]=0;//reset the rotation counter
motor[motorB]=100; //start the motor
while (nMotorEncoder[motorB]<1080) { //wait for 3 rotations
}
motor[motorB]=0;
```

The drawback of this code snippet is that upon reaching 1080 degree rotation, the motor will abruptly stop, changing from 100 power to 0 power. In real life, it means that momentum will carry the robot farther than we intended, slightly overshooting the target distance. A smarter algorithm would slow the motor down when it is approaching the target value.

RobotC has such a smart algorithm built-in (it is a variation of so-called PID algorithm, which is the industry standard algorithm for controlling any system using feedback values — everything from toy cars to nuclear reactors). Here is the program illustrating the use of this algorithm.

```
nMotorEncoder[motorB]=0;//reset the rotation counter
nMotorEncoderTarget[motorB] = 1080; // Set the target of 3 rotations
motor[motorB]=100;                  //start the motor
while(nMotorRunState[motorB]==runStateRunning){
  // wait for the motor to stop running
}
motor[motorB]=0;
```

Command `nMotorEncoderTarget[motorB] = 1080;` sets the target value for the encoder and activates the PID motor control algorithm. However, it does not by itself start the motor, so we do it manually, using `motor[motorB]=100;` command. What setting the target value does is that it will change the speed, slowing the motor down when it is getting close to the specified value (and reversing the direction if some outside force makes the motor overshoot the target).

The final while loop is there to wait for the PID algorithm to finish its work in stopping the motor. It uses expression `nMotorRunState[]` which gives the status of the motor. Possible values are `runStateRunning`, `runStateIdle` and `runStateHoldPosition`. The first two are self-explanatory; the last one we will not be using.

The Natural Language commands described in Section **??** use the RobotC PID algorithm.

9.7. Synchronizing the motors

Motor encoders can be used for one more purpose: making sure that two motors rotate exactly in sync, at the same speed. This is a very common need when the two motors are driving left and right wheels of a robot: just setting power to each motor at 100 does not necessarily gives motion in a straight line, as the motors might be slightly different.

To set up the motors to be synchronized, you need to specify one of them as master (which you will control directly) and the other one as the slave; RobotC will use encoders to control the rotation speed of the slave motor, based on the speed of the master. The following code snippet illustrates it:

```
nSyncedMotors = synchBC; // Motor B is  master,  C is  slave
nSyncedTurnRatio = 100; // motorC rotates at the same speed as motorB
motor[motorB] = 50; // Motor B moves at 50% power
wait1Msec (1000);
```

If you want to use another pair of motors, just replace letters B and C in `synchBC` as needed. For example, to set motor C as the master and A as the slave, you would use `nSyncedMotors = synchCA;`.

You can have the slave motor move slower than the master motor by changing `nSyncedTurnRatio` parameter. Setting it to 100 means that slave and master move at the same rate; setting it to 50 would make the slave motor rotate at exactly half the speed of the master motor, and setting it to -100 would make the slave motor rotate at exactly the same speed as the master but in the opposite direction. Note that you can not make the slave motor move faster than the master — if you need this, just switch the roles of master and slave motors.

Warning. Note that once you have set up a pair of motors to be synced, they remain synced from this moment on, so starting the master motor will automatically start the slave one, whether you want it or not. If you want to unset the synchronization between two motors, use the command `nSyncedMotors=synchNone;`.

9.8. Joystick control

If you have a joystick (a Logitech USB joystick controller or compatible), you can use it to control your robot. Of course, it requires that during the program execution the robot must be connected to the computer, either by a USB cable or via Bluetooth. Please see RobotC documentation under NXT FUNCTIONS→JOYSTICK CONTROL for details. Note that in order for this feature to work, you need to keep the window `Joystick Control - Simple` open; see NXT FUNCTIONS→JOYSTICK CONTROL→JOYSTICK CONTROLLER STATION in RobotC documentation.

RobotC recently introduced a feature that allows one to configure and use other joystick controllers (such as Xbox360 controller). You can find documentation of this feature online at `http://www.robotc.net/wiki/Custom_Joystick_Controls`.

Chapter 10

Troubleshooting

10.1. Some robot-building hints

- Brace everything! Never have a large part (motor/arm/attachment) that is only connected to the robot at one point — add crossbeams, diagonal supports, braces. The robot should be able to survive violent shaking — if not, it needs strengthening.
- If you are using the robot design with two drive wheels and a caster wheel or ball (which is by far the most common design), make sure that most of the weight is on the drive wheels and not on the caster wheel/ball. If necessary, use a counterweight. Also, the robot should be moving so that the drive wheels are in the front and the caster wheel is at the back.
- Avoid making your robot too large. A large robot is very difficult to turn (especially in a tight spot, or when you are moving along a wall) and is prone to breaking. Compact design is usually the best.

10.2. Clicking brick syndrome

If for some reason the the firmware update was unsuccessful, you may end up with a brick which looks dead – doesn't react to button presses and shows nothing on the display.

To fix this problem, you will need to reset the brick. Disconnect the brick from computer (some authors also recommend disabling Bluetooth on the computer to prevent the computer from connecting to the brick via Bluetooth) and try first these steps:

(1) Remove one of the batteries from the NXT brick, wait for several seconds, then re-insert the battery. See if the brick starts now.

(2) If that doesnt work, remove all batteries. You will notice a little button inside the battery compartment. Push it down for 5 seconds. This will create a "soft" reset. Now reinsert the batteries.

If the above steps didn't work you will need to do a "hard" reset. NXT brick has a reset button hidden at the bottom of one of the peg holes, the one under the USB port. This button can be pressed using a straightened paperclip as shown below:

To do the hard reset, put the batteries back in NXT, and press the reset button using a paperclip for 5 full seconds. Make sure you are holding the button down for at least 5 seconds and release it. It should bring the NXT in so-called SAMBA mode (used for downloading firmware), which is a dead screen and a quiet clicking noise.

If the NXT is clicking before you press the hardware reset button, it will stop clicking after you press the hardware reset button. Continue to press on the hardware reset button for a few seconds longer until it clicks once. Now release the hardware reset button and wait for the NXT to start clicking continuously again (it will take approximately 2-4 seconds).

Now connect the NXT to the computer by a USB cable, and proceed with downloading the firmware as described in Section 1.3.

10.3. Broken screen

A common problem with NXT is the screen stopping working. It seems to be a known manufacturing defect (some connection falling apart). It is theoretically possible to fix by taking NXT apart and re-soldering some contacts (search for instructions online), but you must be quite good at soldering to do this. It would also void warranty on your NXT. Another option is calling LEGO group and asking them to replace the NXT; they have been known do it even if the NXT is past the warranty period.

But you can also use the NXT perfectly well with non-working screen. To see messages on the screen, you can use the `NXT Remote Screen` window available in Robot→Debugger Windows.

10.4. Debugging a program

If a program does not work as expected, try the tips below to help you figure out what is wrong.

- **Check for hardware problems.** If a robot is veering to the side instead of going straight, it might be caused by a wheel rubbing on something. If a sensor is not properly working, make sure that the cable connecting it to NXT is not loose. Also, check the tips on robot-building in Section 10.1.
- **Test the sensors.** Make sure the threshold values for light and ultrasonic sensros are correct. You can check the sensor readings as described in Section 6.4.
- **Insert stops in the program.** In order to better see at what place in the program something goes wrong, insert stops. Placing in the program the line

```
PlaySound(soundBeepBeep); wait1Msec(3000);
```

would play a sound to signal that you have reached this place in the program and then pause the robot for 3 seconds, giving you time to note the robot location and direction. You can also print some sensor readings, variable values, or other info to NXT display.

You can use sounds for feedback, for example playing one tone when the sensor sees black and another when it sees white. Just remember that the sound requests are queued.

- **Use debugger.** For long programs or more serious problems, you can use RobotC built-in debugger (the `Program Debug` window that appears after you compile and download a program). This requires that the robot be connected to the computer (via USB cable or Bluetooth), but provides a variety of tools — such as being able to execute the program one step at a time, analyze the values of all the variables, and more. Please refer to RobotC documentation under ROBOTC Debugger→Debugging.

Chapter 11

List of Commonly Used Commands

In this chapter, we list some commonly used commands and constructions of RobotC.

11.1. Control structure

If statement:

```
if (condition) {
  some commands
} else {
  some other commands
}
```

While loop

```
while (condition) {
  some commands
}
```

11.2. Basic commands

Function	Description	Arguments
`wait1Msec(time)`	Wait given time	`time` (in milliseconds)
`motor[motorname]=power;`	turn on the given motor at given power	`motorname`: motor, e.g. `motorA` `power`: power (integer between -100 and 100)
`playSound(sound)`	play a sound	`sound`: one of the listed sounds, e.g. `soundBeepBeep`. See Section 9.1 for the list.

11.3. Natural Language commands

These commands are only available after some initial setup. See Section 7.1 for details.

Function	Description	Arguments
`NLRotateMotor(motor, deg, power)`	Rotates one motor given number of degrees	`motor`: motor to be rotated (e.g. `motorA`) `deg`: number of degrees (can be negative) `power` (optional)
`NLStartForward(power)` `NLStartBackward(power)`	Starts both drive motors going forward/backward. You will need another command to stop the motors	power (optional)
`NLStopMotors()`	Stops both drive motors	
`NLGoForward(dist, power)` `NLGoBackward(dist, power)`	Move forward/backward for specified distance, using drive motors	`dist`: distance (in MOVEUNITS; 1 MOVEUNIT $\approx$ 1 cm) `power` (optional)
`NLGoForwardTime(time, power)` `NLGoBackwardTime(time, power)`	Move forward/backward for specified time, using drive motors	`time`: time (in seconds) `power` (optional)
`NLTurnLeft(units, power)` `NLTurnRight(units, power)`	Turn left/right in place, rotating wheels in opposite directions	`units`: number of units to turn. 1 unit $\approx$ 1 degree `power` (optional)

The parameter `power` in all the commands sets the power of the motors. It must be an integer between 0-100 (do not use negative values). It is optional: it can be omitted, in which case the default value of 75 is used. For example, writing `NLStartForward()` is the same as `NLStartForward(75)`.