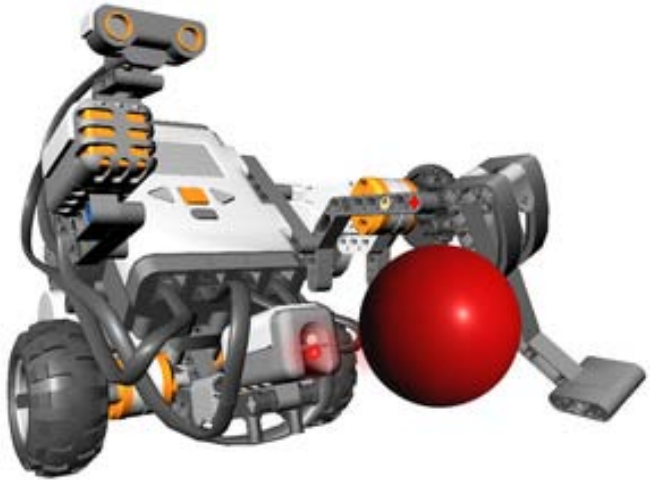


RobotC Programming for LEGO Mindstorms NXT

SOURCES:

Carnegie Mellon
Dacta
Lego
Timothy Friez

Miha Štajdohar
miha.stajdohar@fri.uni-lj.si

Loading Firmware

- Robots require “Operating Systems” to run properly
- ROBOTC has it’s own Operating System, called Firmware.
- The firmware must be loaded onto the robot controller before.

Downloading the Firmware

- Three Steps to downloading firmware
  - 1. Make sure your platform type is correct
    - ROBOTC – Robot Menu – Platform Type – FRC
  - 2. Make sure your robot is powered on and in Programming mode.
    - On the FRC, hold the “PROG” button until two LEDs become orange.
  - 3. Download the firmware
    - ROBOTC – Robot Menu – Download Firmware – “FRC VM 0724.hex”

Programming in RobotC for practical robotic applications

ROBOTC Fundamentals

- Every program starts with a “task main”
  - Exception: Competition Template programming
    - The “task main” is taken care of you behind the scenes.

```
task main()
{

}
```

Example RobotC Programs: forward and spin for motors

```
void forward() {
    motor[motorA] = 100;
    motor[motorB] = 100;
}

void spin() {
    motor[motorA] = 100;
    motor[motorB] = -100;
}
```

Example RobotC Program - BEHAVIOR

```
task main() {
  SensorType[S4] = sensorSONAR;
  forward();
  while(true) {
     if (SensorValue[S4] < 25)
  spin();
     else forward();
  }
}
```

Explain the role of behaviors in robot programming

Syntax and statements

- Programs consist of a number of statements.
- Statements tell the robot controller what to do.
- **Motor[port1] = 127;** is a statement
- All statements end with semicolons "**;**"
- Proper syntax (semicolons, brackets, etc) is very important.

Detailed programming is necessary

- Robots are dumb by nature.
  - They will only do what they are told.

- A robot will run forever unless told to stop.

- `Motor[port1] = 127;`
- `Motor[port2] = 127;`

- Robots will also only execute code once unless told otherwise. (a loop!)

Comments in ROBOTC

- Comments are very important for memory and understanding
- // - Single Line Comment

- /*
- Multiple line comments
- */

Code conventions. Lower Case and Uppercase.

1. Text written as part of a program is called **code**.
2. You type code just like you type normal text.

3. Keep in mind that capitalization is important to the computer.

4. Replacing a lowercase letter with a capital letter (or a capital letter with a lowercase letter) will cause the robot to become confused.

```
1  Task main()
2  {
3
4      motor[port3] = 127;
5      wait1Msec(3000);
6
7  }
```

Capitalization
Capitalization (paying attention to UPPERCASE vs. lowercase) is important in ROBOTC.

If you capitalize the 'T' in task, ROBOTC no longer recognizes this command.

Code COLOR

1. As you type, ROBOTC will try to help you out by coloring the words it recognizes.

2. If a word appears in a different color, it means ROBOTC recognizes it as an important word in the programming language.

```
1  task main()
2  {
3
4      motor[port3] = 127;
5      wait1Msec(3000);
6
```

Code coloring
ROBOTC automatically colors key words that it recognizes.

Compare this correctly-capitalized "task" command with the incorrectly-capitalized version in the previous example. The correct one is recognized as a command and turns blue.

```
1  Task main()
2  {
3
4      motor[port3] = 127;
5      wait1Msec(3000);
```

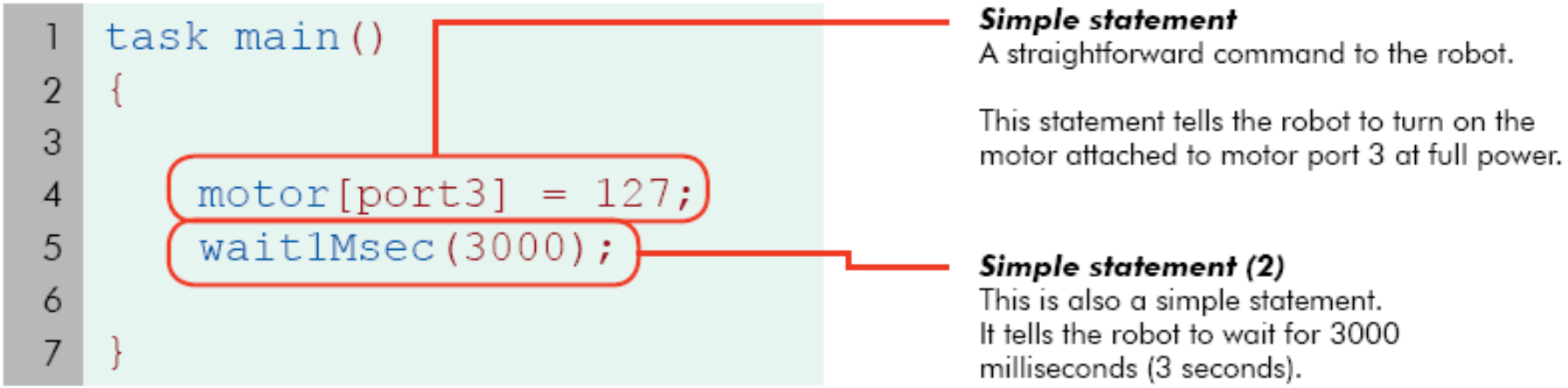
Statements

1. Statements are instructions for the robot.

2. The most basic kind of statement in ROBOTC simply gives a command to the robot.

3. The **motor[port3] = 127;** statement in the sample program you downloaded is a simple statement that gives a command.

4. It instructs the motor plugged into Motor Port 3 to turn on at full power.



Order the Statements

1. Statements are run in order as quickly as the robot is able to reach them.
2. Running this program on the robot turns the motor on, then waits for 3000 milliseconds (3 seconds) with the motor still running, and then ends.

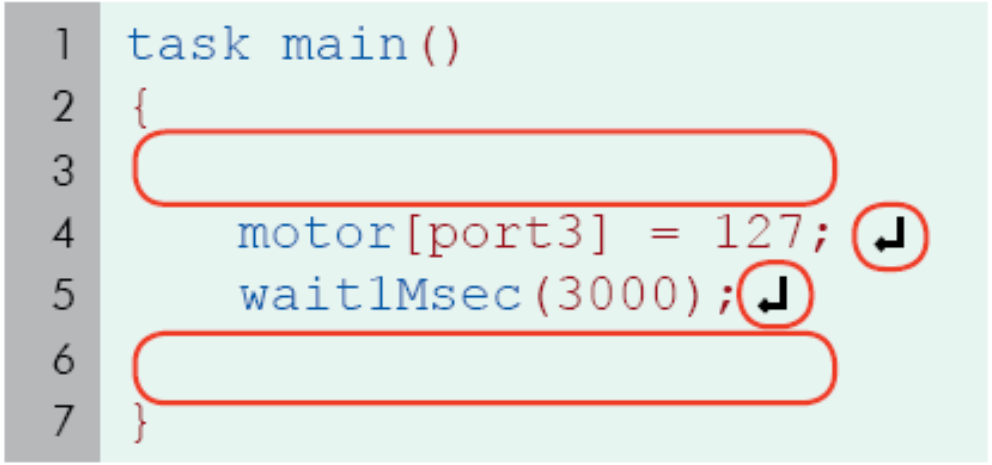
```
1  task main()
2  {
3
4    1st motor[port3] = 127;
5    2nd wait1Msec(3000);
6
7  } End
```

Sequence
Statements run in English reading order (left-to-right, top-to-bottom). As soon as a command is complete, the next one runs. These two statements cause the motors to turn on *(1st command)*. The robot then immediately begins a three second wait *(2nd command)* while the motors remain on.

End
When the program runs out of statements and reaches the } symbol in `task main`, all motors stop and the program ends.

RobotC Rules: role of semicolon

1. *How did ROBOTC know that* **motor[port3]= 127** and **wait1msec[3000]** *were two separate commands*.
2. Was it because they appeared on two different lines?



Whitespace
Spaces, tabs, and line breaks are generally unimportant to ROBOTC and the robot.

They are sometimes needed to separate words in multi-word commands, but are otherwise ignored by the machine.

•No.
•Spaces and line breaks in ROBOTC are only used to separate words from each other in multi-word commands.
•Spaces, tabs, and lines don't affect the way a program is interpreted by the machine.

ROBOTC Rules: the role of spacing

1. So why ARE they on separate lines?
   - For the programmer.
2. Programming languages are designed for humans and machines to communicate.
3. Using spaces, tabs, and lines helps human programmers read the code more easily.
4. Making good use of spacing in your program is a very good habit for your own sake.

```
1 task main(){motor[port3]
2 =127;wait1Msec(3000);}
```

No Whitespace

To ROBOTC, this program is the same as the last one. To the human programmer, however, this is close to gibberish.

Whitespace is used to make programs readable to humans.

Punctuation! Semicolons!

1. But what about ROBOTC?
2. How DID it know where one statement ended and the other began?
3. It knew because of the semicolon (;) at the end of each line.
4. **Every statement ends with a semicolon**. It's like the period at the end of a sentence.

```
1  task main()
2  {
3
4      motor[port3] = 127;
5      wait1Msec(3000);
6
7  }
```

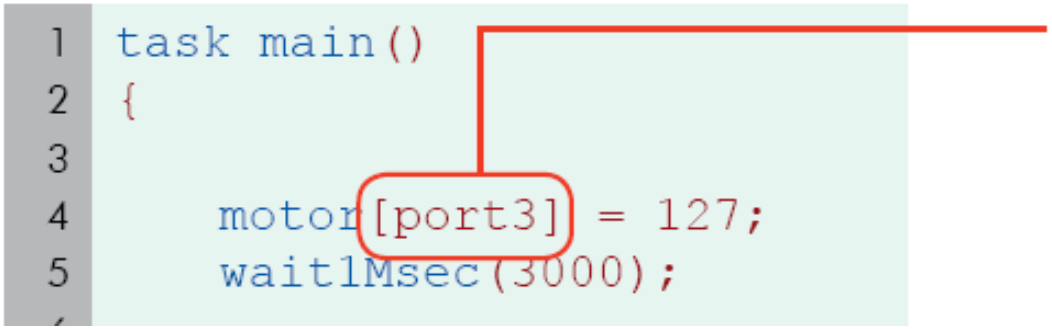
Semicolons
Like periods in an English sentence, semicolons mark the end of every ROBOTC statement.

Punctuation Pairs: Matching Parentheses

1. Punctuation pairs, like the parentheses and square brackets in these two statements, are used to mark off special areas of code.

2. Every punctuation pair consists of an opening punctuation mark and a closing punctuation mark.

3. The punctuation pair designates the area *between them* as having special meaning to the command that they are part of.



Punctuation pair: Square brackets [ ]
The code written between the square brackets of the `motor` command indicates which motor the command should use. In this case, it is the motor on port 3.

Punctuation Pairs: Square Brackets

```
1  task main()
2  {
3
4      motor[port3] = 127;
5      wait1Msec(3000);
6
7  }
```

Punctuation pair: Parentheses ( )
The code written between the parentheses of the `wait1Msec` command tell it how many milliseconds to wait before starting a new command. In this case, it waits 3000 milliseconds, or three seconds.

1. Different commands make use of different kinds of paired punctuation.

2. The **motor** command uses square brackets and the **wait1Msec** command uses parentheses.

3. This is just the way the commands are set up.

- You will have to remember to use the right punctuation with the right commands or plan.

Control Structures

1. *Simple statements do the work in ROBOTC, but control structures do the thinking.*

2. Control structures (or control statements) are pieces of code that control the flow of the program's commands, rather than issue direct orders to the robot.

```
1  task main()
2  {
3
4      motor[port3] = 127;
5      wait1Msec(3000);
6
7  }
```

Control structure: `task main`
The control structure `task main` directs the program to the main body of the code. When you click the ***Start*** button in ROBOTC or turn on the robot, the program immediately goes to `task main` and runs the code it finds there.

The left and right curly braces { } belong to the `task main` structure. They surround the commands which will be run in the program.

1. One important control structure is **task main**.

2. Every ROBOTC program includes a special section called **task main**.

3. This control structure determines which code the robot will run as part of the main program.

Control Structures

```
while(SensorValue(touchSensor) == 0)
{
    motor[port3] = 127;
    motor[port2] = 127;
}
```

Control structure: **`while`** ***loop***
The while loop repeats the code between its curly braces `{ }` as long as certain conditions are met.

Normally, statements run only once. But with a **`while`** loop, they can be told to repeat over and over for as long as you want!

Comments: write your code incrementally

1. Comments are text that the program ignores.

2. A comment can contain notes, messages, and symbols that may help a human, but would be meaningless to the robot.

3. ROBOTC simply skips over them. Comments appear in green in ROBOTC.

```
1  // Motor port 3 forward with 100% power
2
3  task main()
4  {
5
6    /*
7      Port 3 forward with 100% power
8      Do this for 3 seconds
9    */
10
11     motor[port3] = 127;
12     wait1Msec(3000);
13
14 }
```

Comments: // Single line
Any section of text that follows a // (two forward slash characters) on a line is considered to be a comment. Any text to the left of the // is treated as normal code.

Comments: /* Any length */
A comment can be created in ROBOTC using another type of paired punctuation, which starts with /* and ends with */ This type of comment can span multiple lines, so be sure to include both the opening and closing marks!

Motor Commands

- Motor[port] = speed;
- Motor – tells the controller we want to use a motor (PWM) port
- [port] – tells the controller what port to use
  - [port1], [port2], [port3], …, [port16]
- = speed; - assigns a value to the motor telling it what speed to travel
  - 127 is full forward, -127 is full reverse,0 is stop

Using Joysticks

- To control your robot using a joystick axis or a button, use these functions

- frcRF[port] – gets the value from a joystick axis
  - [port] – p1_y, p1_x, p1_wheel, … p2_x,…p3_x, etc.

- frcOIJoystickButtons[port]
  - [port] – oiButtonPort1Button1 ***through*** oiButtonPort4Button4

Slowing motors down

- ROBOTC lets you use math functions to control your motors speed
- Motor[port1] = (127 / 2);
  - This would actually set the motor to 63.5… rounded down to 63.
- You can use this when accessing sensor values and joystick values also
- Motor[port1] = (frcRF[p1_y] / 2)
  - This joystick input is now divided by 2 before being sent to the motor.

Using Loops

- A Loop will help a portion of your code execute repeatedly, until told to stop.

```
while(true == true)
{
motor[port1] = frcRF[p1_y];
motor[port2] = frcRF[p2_y];
}
```

- This code will cause the motors to respond to the joystick values forever.
  - The loop reads (while true is equal to true)

Analog Sensors

- Very easy to read analog sensor values
  - SensorValue[port]
  - [port] = in1,in2,in3 … in16

```
while(SensorValue[in1] < 10)
{
motor[port1] = frcRF[p1_y];
motor[port2] = frcRF[p2_y];
}
```

- This program will run until an analog sensor on port 1 sees a value of less than 10.
- Analog sensor values can range from 0 to 1023

Digital Sensors

- Digital sensors are a little more complex
  - First you have to decide if you want the sensor port to be an input or output
  - frcDigitalIODirection[port] = condition;
    - port – pio1, pio2, pio3, … pio18
    - condition – dirInput or dirOutput
  - Second you have read or assign the value
  - frcDigitalIOValue[port]
    - If an input, this will read the value (either a 1 or 0)
    - If an output, you can assign the value using a 1 or 0.
      - frcDigitalIOValue[port] = 1; // turns on the digital output port

Relays

- Relays are easy to control. There are 4 states possible for each relay:
  - frcRelay[port] = condition;
  - [port] – relay1, relay2, relay3 … relay8
  - conditions:
    - relayFwd – Sets the relay to forward
    - relayRvs – Sets the relay to reverse
    - relayOff – Sets the relay to a off (float) state
    - relayBrake – Sets the relay to a braking (short) state