Programming robots with NXC

Eric Martin, Tim Lambert and Claude Sammut

School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
Acknowledgments

This presentation borrows material from three sources:

- Tutorials on NQC and NXC by Mark Overmars, entitled *Programming Lego® Robots using NQC*, which is available at http://www.cs.uul.nl/people/markov/lego/index.html
Hank the bumper car

- We will test our first programs and meet our first challenges with a robot called Hank.
- Hank is a bumper car equipped with two touch sensors.
- Build Hank following the instructions in your kit.
Launch the Bricx command center.

Type in, **compile**, and run the following program, after predicting what it is meant to do.

```c
task main()
{
    OnFwd(OUT_A, 50);
    OnFwd(OUT_C, 50);
    Wait(4000);
    OnRev(OUT_AC, 50);
    Wait(4000);
    Off(OUT_AC);
}
```

The **layout** of the program, and in particular, the indentation and the position of the curly braces, does not matter for the program to compile, but it is essential to achieve maximum readability.
Tasks and statements

- Programs in NXC consist of tasks.
- Our first program has just one task, named main.
- Each program has a main task, possibly with other tasks.
- A program starts by executing its main task.
- A task consists of a number of commands, also called statements.
- Tasks in the body of a task are surrounded by curly braces.
- Each statement ends in a semicolon.
- So a task looks in general as follows:

```c
task task_name()
{
    statement1;
    statement2;
    ...
}
```
A few statements (1)

The body of the main task of our first program consists of 6 statements:

OnFwd(OUT_A, 50) Tells the robot to start output A. The motor connected to output A will spin forward with 50% power.

OnFwd(OUT_C, 50) Same statement, but for output C. After both statements are executed, the robot is moving forwards at half speed.

Wait(4000) Tells the program to wait for 4 seconds. The argument to Wait, i.e., the number between the parentheses, gives the number of “ticks.” Each tick is one millisecond (1/1000 of a second). So for 4 seconds, the program does do nothing and the robot keeps moving.
OnRev(OUT_AC, 50) Tells the robot to spin both motors in reverse direction at 50% power. Note how OUT_AC allows combining two statements into one.

Wait(4000) Tells the program to wait for 4 seconds again.

Off(OUT_AC) Tells the robot to switch both motors off.

So the whole program moves both motors forwards for 4 seconds, then backwards for 4 seconds, and finally switches them off.
Making turns

Type in, compile, and run the following program, after predicting what it is meant to do.

```c
#define POWER 50
#define MOVE_TIME 1000
#define TURN_TIME 850

task main()
{
    OnFwd(OUT_AC, POWER);
    Wait(MOVE_TIME);
    OnRev(OUT_C, POWER);
    Wait(TURN_TIME);
    Off(OUT_AC);
}
```
The first three lines of the previous program define three constants. Constants can be used throughout the program. Defining constants has two purposes:
- it makes the program more readable;
- it makes it easier to change the values.

You might have to change the value of TURN_TIME to achieve a precise 90 degree turn.
Repeating commands

Type in, compile, and run the following program, after predicting what it is meant to do.

```c
#define POWER 50
#define MOVE_TIME 1000
#define TURN_TIME 850

void main()
{
    repeat(4)
    {
        OnFwd(OUT_AC, POWER);
        Wait(MOVE_TIME);
        OnRev(OUT_C, POWER);
        Wait(TURN_TIME);
    }
    Off(OUT_AC);
}
```
The number provided as an argument to `repeat` indicates how many times the statements in the body of the `repeat` loop must be executed.

The robot should move forward and make turns four times, hence should drive in a square.

Note that the statements that make up the body of the `repeat` statement are enclosed between curly braces, just like the statements in the body of a task.

Note the layout of the program—the different levels of indenting and the position of the curly braces—that maximises readability.
The repeat command (2)

- Repeat commands can be nested.
- The next program makes the robot run 3 times in a square.

```c
#define POWER 50
#define MOVE_TIME 1000
#define TURN_TIME 850

task main() {
    repeat(3) {
        repeat(4) {
            repeat(4) {
                OnFwd(OUT_AC, POWER);
                Wait(MOVE_TIME);
                OnRev(OUT_C, POWER);
                Wait(TURN_TIME);
            }
        }
        Off(OUT_AC);
    }
}
```

- Again, note how the different levels of indenting and the position of the curly braces maximise readability.
Comments make a program even more readable.

There are two kinds of comments:

// What comes to the right of // on the same line is ignored.
/* ... */ Everything between /* and */ is ignored.
This kind of comment can span multiple lines.

Comments should be informative, but not redundant. For instance, there is no point to comment the statement Off(OUT_AC); with // Turn motors off
Indeed, the statement is clear enough by itself.

As an example, the previous program can be enhanced with comments as shown next.
/* Driving 3 times in a square
    by Mark Overmars, modified by Claude Sammut */
#define POWER 50
#define MOVE_TIME 1000 // Time for a straight move
#define TURN_TIME 850 // Time for turning 90 degrees

task main()
{
    repeat (3) {
        repeat (4) {
            OnFwd(OUT_AC, POWER);
            Wait(MOVE_TIME);
            OnRev(OUT_C, POWER);
            Wait(TURN_TIME);
        }
    }
    Off(OUT_AC);
}
Type in, compile, and run the following program, after predicting what it is meant to do.

```c
#define POWER 50
#define TURN_TIME 850
int move_time;
task main()
{
    move_time = 200;
    repeat (50)
    {
        OnFwd(OUT_AC, POWER);
        Wait(move_time);
        OnRev(OUT_C, POWER);
        Wait(TURN_TIME);
        move_time += 50;
    }
    Off(OUT_AC);
}
```
Defining and initialising variables

- A variable is defined by typing the keyword `int`, followed by the name of the variable.
- `int` means `integer`.
- A name is a string of (upper case or lower case) letters, underscore or digits that does not start with a digit.
- A usual convention is to use no upper case letter for variable names, and no lower case letters for constants. Still this is not a requirement.
- To maximise readability, variables should be given meaningful names.
- The statement `move_time = 200;` initialises `move_time` to 200.
- Alternatively, `move_time` could have been defined and initialised at the same time with the statement `move_time = 200;`
- Also, many variables can be defined (and possibly initialised) in one statement: they just have to be separated by commas.
The value of move_time is changed 50 times, every time the repeat loop is executed.

When the repeat loop begins execution, the value of move_time is equal to 200.

At the end of each execution of the repeat loop, the value is incremented by 50.

So the robot will move forward for 200 ticks (how much is that in seconds?), turn right, move forward for 250 ticks, turn right, move forward for 300 ticks, turn right, etc.

For how long will the robot move forward the last time the repeat loop is executed?
Besides adding values to a variable, we can also multiply, subtract and divide a variable with a number using, respectively, `*=` `-=`, and `/=`.

Division rounds the result to the nearest integer. For instance, 8 divided by 3 yields 2.

We can also add one variable to the other, and write down more complicated expressions. For instance:

```c
int a;
int b, c;
task main()
{
    a = 10;
    b = 20 * 5;
    c = b;
    c /= a;
    c -= 5;
    a = 10 * (c + 3); // What is the value of a now?
}
```
Unpredictable behaviour

Type in, compile, and run the following program, after predicting what it is meant to do.

```
#define POWER 50
int move_time, turn_time;
task main()
{
    while (true)
    {
        move_time = Random(600);
        turn_time = Random(400);
        OnFwd(OUT_AC, POWER);
        Wait(move_time);
        OnRev(OUT_A, POWER);
        Wait(turn_time);
    }
}
```
We can generate random numbers to make the robot react in an unpredictable way.

The previous program defines two variables, and then assigns random numbers to them.

_random(600)_ generates a random number between 0 and 600 (included), while _random(400)_ generates a random number between 0 and 400 (included).

The previous program also introduces a construct that defines a second kind of loop (besides repeat), using the while keyword.

The while statement keeps executing the statements in its body as long as the condition provided as an argument is true.

The keyword true always evaluates to true, so the statements in the body of the loop are repeated forever, creating an infinite loop.
Type in, compile, and run the following program, after predicting what it is meant to do.

```c
#define POWER 50
#define MOVE_TIME 1000
#define TURN_TIME 850

void main()
{
    while(true)
    {
        OnFwd(OUT_AC, POWER);
        Wait(MOVE_TIME);
        if (Random() >= 0)
        {
            OnRev(OUT_C, POWER);
        }
        else
        {
            OnRev(OUT_A, POWER);
        }
        Wait(TURN_TIME);
    }
}
```
The if/else statement

- The if/else statement is used to have one part of the program execute in particular situations, and another part of the program execute in the other situations.

- The previous program makes the robot turn left or right after driving along a straight line. The decision to turn left or right is made at random.

- The test Random() \( \geq 0 \) generates an arbitrary integer that is either positive or negative. This statement is like flipping a coin.

- The else statement is optional. If the condition of the if test evaluates to false (i.e., 0), then the body of the if statement is skipped and execution resumes afterwards.
Comparisons

- **Distinguish between**
  - \( a == b \), which tests whether \( a \) and \( b \) have equal values, evaluates to true if that is the case and to false otherwise, and
  - \( a = b \), which assigns the value of \( b \) to \( a \).

- **To compare values**, you can use other **comparison operators**:
  - \( a \neq b \) evaluates to true iff \( a \) and \( b \) have distinct values;
  - \( a < b \) evaluates to true iff the value of \( a \) is strictly smaller than the value of \( b \);
  - \( a \leq b \) evaluates to true iff the value of \( a \) is smaller than or equal to the value of \( b \);
  - \( a > b \) evaluates to true iff the value of \( a \) is strictly greater than the value of \( b \);
  - \( a \geq b \) evaluates to true iff the value of \( a \) is greater than or equal to the value of \( b \).
Simple comparisons can be combined into more complex conditions using one or more **Boolean operator**.

- `&&` whose meaning is *and*;
- `||` whose meaning is *or*;
- `!` whose meaning is *not*.

Here are some examples of conditions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true</code></td>
<td>always true</td>
</tr>
<tr>
<td><code>false</code></td>
<td>never true</td>
</tr>
<tr>
<td><code>(a &gt;= 5) &amp;&amp; (a &lt;= 10)</code></td>
<td>true iff <code>a</code> lies between 5 and 10</td>
</tr>
<tr>
<td>`(a == 10)</td>
<td></td>
</tr>
<tr>
<td><code>!(a &gt; 5) &amp;&amp; (a &lt; 9)</code></td>
<td>true iff <code>a</code> is distinct to 6, 7 or 8</td>
</tr>
</tbody>
</table>

Parentheses are used to give **priority** to some operators over others.
Using touch sensors

Type in, compile, and run the following program, after predicting what it is meant to do.

```c
task main()
{
    SetSensor(IN_1, SENSOR_TOUCH);
    OnFwd(OUT_AC, 50);
    until (SENSOR_1 == 1);
    Off(OUT_AC);
}
```
The first statement in the previous program tells the robot what type of sensor is connected to which port.

IN_1 is the number of the input to which the sensor is connected.

The other two sensor inputs are called IN_2 and IN_3.

Of course, SENSOR_TOUCH indicates that a touch sensor is being used; for a light sensor we would write SENSOR_LIGHT.

The third statement waits until the condition SENSOR_1 == 1—which could be simplified to SENSOR_1—evaluates to true, indicating that the sensor is being pressed.

As long as the touch sensor connected to port 1 is not pressed, SENSOR_1 evaluates to 0 (i.e., to false).
First programming exercise

- Write an NXC program for Hank to avoid obstacles.
- Whenever the robot hits an object, it should move back a bit, make a turn, and then continue.
- The robot should turn left if the right bumper has first hit the obstacle, and right otherwise, as shown in the movie below.
Solution (1)

```c
#define POWER 50
#define BACK_TIME 500
#define TURN_TIME 800

void main()
{
    SetSensor(IN_1, SENSOR_TOUCH);
    SetSensor(IN_4, SENSOR_TOUCH);
    OnFwd(OUT_AC, POWER);
}
```

while (true)
{
    if (SENSOR_1) {
        OnRev(OUT_AC, POWER);
        Wait(BACK_TIME);
        OnFwd(OUT_A, POWER);
        Wait(TURN_TIME);
        OnFwd(OUT_C, POWER);
    }
    if (SENSOR_4) {
        OnRev(OUT_AC, POWER);
        Wait(BACK_TIME);
        OnFwd(OUT_C, POWER);
        Wait(TURN_TIME);
        OnFwd(OUT_A, POWER);
    }
}
}
Programs with many tasks (1)

Type in, compile, and run the following program, after predicting what it is meant to do.

```c
#define POWER 50

void main() {
    SetSensor(IN_1, SENSOR_TOUCH);
    start check_sensors;
    start move_square;
}

void move_square() {
    while (true) {
        OnFwd(OUT_AC, POWER);
        Wait(1000);
        OnRev(OUT_C, POWER);
        Wait(850);
    }
}
```

task check_sensors()
{
    while (true)
    {
        if (SENSOR_1 == 1)
        {
            stop move_square;
            OnRev(OUT_AC, POWER);
            Wait(500);
            OnFwd(OUT_A, POWER);
            Wait(850);
            start move_square;
        }
    }
}
Starting and stopping tasks

- An NXC program consists of at most 10 tasks.
- Many tasks can run simultaneously.
- Any task `some_task` except `main` will only be executed when a running task asks for `some_task` to start execution using the `start` command.
- A running task can also stop another running task by using the `stop` command.
- A task that has been stopped can be restarted again, but it will start from the beginning; not from the place where it was stopped.
Second programming exercise

- Imagine that Hank is a living creature which is attacked by a predator. If one of Hank’s bumpers is hit again soon after Hank has been hit by the predator, then it is better to let Hank back up further: *instead of finishing a sequence of actions triggered by a hit, it is better to start that kind of sequence again.*

- Two solutions are possible. Modify your program to improve Hank’s behaviour.
The following programs are very similar and at first sight might both be thought to be valid.

The time during which a touch sensor returns a value of 1 is much larger than the execution time of a command. This is a problem with the first program, but not with the second one.

Remove the comments in the programs to make use of the variable `count`. This variable can have its value displayed to indicate how many times the test that one of the bumpers has hit an obstacle succeeds.
An incorrect solution (1)

```c
#define POWER 50
#define BACK_TIME 500
#define TURN_TIME 800

task main() {
    // int count = 0;
    SetSensor(IN_1, SENSOR_TOUCH);
    SetSensor(IN_4, SENSOR_TOUCH);
    OnFwd(OUT_AC, POWER);

    while (true) {
        if (SENSOR_1 || SENSOR_4) {
            // count++;
            stop back_up;
            start back_up;
        }
    }
}
```
task back_up() {
    if (SENSOR_1) {
        OnRev(OUT_AC, POWER);
        Wait(BACK_TIME);
        OnFwd(OUT_A, POWER);
        Wait(TURN_TIME);
        OnFwd(OUT_C, POWER);
    }
    if (SENSOR_4) {
        OnRev(OUT_AC, POWER);
        Wait(BACK_TIME);
        OnFwd(OUT_C, POWER);
        Wait(TURN_TIME);
        OnFwd(OUT_A, POWER);
    }
}

A correct solution (1)

#define POWER 50
#define BACK_TIME 500
#define TURN_TIME 800

task main()
{
    // int count = 0;
    SetSensor(IN_1, SENSOR_TOUCH);
    SetSensor(IN_4, SENSOR_TOUCH);
    OnFwd(OUT_AC, POWER);
while (true) {
    if (SENSOR_1) {
        // count++;
        stop back_up_1;
        stop back_up_4;
        start back_up_1;
    }
    if (SENSOR_4) {
        // count++;
        stop back_up_1;
        stop back_up_4;
        start back_up_4;
    }
}
A correct solution (3)

#define POWER 50

task back_up_1() {
    OnRev(OUT_AC, POWER);
    Wait(BACK_TIME);
    OnFwd(OUT_A, POWER);
    Wait(TURN_TIME);
    OnFwd(OUT_C, POWER);
}

task back_up_4() {
    OnRev(OUT_AC, POWER);
    Wait(BACK_TIME);
    OnFwd(OUT_C, POWER);
    Wait(TURN_TIME);
    OnFwd(OUT_A, POWER);
}
A wrong program (1)

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

```
#include "NXCDefs.h"

task main()
{
    SetSensor(IN_1,SENSOR_TOUCH);
    Precedes(check_sensors, submain);
}
```
A wrong program (2)

#define POWER 75

task check_sensors() {
    while (true) {
        if (SENSOR_1 == 1) {
            OnRev(OUT_AC, POWER);
            Wait(500);
            OnFwd(OUT_A, POWER);
            Wait(850);
            OnFwd(OUT_C, POWER);
        }
    }
}

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

Tasks interfere

- If the robot touches something while it is turning, it starts going back, but sometimes it will move forwards again, hitting the obstacle.
- This happens if, after the check sensors task puts the motors in reverse, the submain task finishes the wait at the end of the loop and then in forward. The check sensors task is waiting to finish reversing and isn’t even checking the touch sensor, so the robot rams the obstacle.
Concurrency occurs everywhere:

- Databases
- Operating Systems
- Circuit Design
Contention for Resources
Semaphores

- When you stop and restart a task, it starts at the beginning.
- OK for small tasks, but we really should stop and resume at the same place in the task.
- One way to assure that happens: use a semaphore
- Semaphore: a global variable accessed by both tasks
  - Semaphore = 0 means no task is driving motors
  - Semaphore = 1 means a task is driving motors

When a task wants to use the motors, execute following code:

```c
until (semaphore == 0);
semaphore = 1;
// Use the motors
semaphore = 0;
```
int sem;

task main()
{
    sem = 0;
    SetSensor(SENSOR_1,SENSOR_TOUCH);
    start check_sensors;
    start move_square;
}
#define POWER 50

task move_square() {
    while (true) {
        until (sem == 0); sem = 1;
        OnFwd(OUT_AC, POWER);
        sem = 0;
        Wait(1000);
        until (sem == 0); sem = 1;
        OnRev(OUT_C, POWER);
        sem = 0;
        Wait(850);
    }
}
task check_sensors()
{
    while (true)
    {
        if (SENSOR_1 == 1)
        {
            until (sem == 0); sem = 1;
            OnRev(OUT_AC, POWER); Wait(500);
            OnFwd(OUT_A, POWER); Wait(850);
            sem = 0;
        }
    }
}
A mutex is a semaphore built into NXC. You use Acquire to acquire exclusive use of the resource, and Release when you are finished.

```c
#include "NXCDefs.h"
define POWER 75

mutex moveMutex;

task main()
{
    SetSensor(IN_1,SENSOR_TOUCH);
    Precedes(check_sensors, move_square);
}
```
task move_square() {
    while (true) {
        Acquire(moveMutex);
        OnFwd(OUT_AC, POWER); Wait(1000);
        OnRev(OUT_C, POWER); Wait(850);
        Release(moveMutex);
    }
}

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

- The do statement is a looping construct very similar to the while statement.
- It has the form:

  ```
  do {
    statements;
  } while (condition);
  ```
- The statements in the body of the do loop are executed as long as the condition evaluates to true. In a while statement, the condition is tested before executing the statements, while in the do statement the condition is tested at the end.
- Hence with a while statement, the statements in the body of the loop might never be executed, whereas with a do statement, they are executed at least once.
Moving around randomly

Type in, compile, and run the following program, after predicting what it is meant to do.

```
#define POWER 50
int move_time, turn_time, total_time;
task main() {
    total_time = 0;
    do {
        move_time = Random(100);
        turn_time = Random(100);
        OnFwd(OUT_AC, POWER);
        Wait(move_time);
        OnRev(OUT_C, POWER);
        Wait(turn_time);
        total_time += move_time;
        total_time += turn_time;
    } while (total_time < 2000);
    Off(OUT_AC);
}
```
Timers

- The NXT has four built-in timers, numbered from 0 to 3.
- A timer ticks in increments of 1/1000 of a second.
- The value of a timer can be reset with the command `ClearTimer()`, providing the timer’s number as argument.
- The current value of a timer with the command `Timer()`, providing the timer’s number as argument.

**Exercise**: change the previous program by making use of a timer, with no need for the variables `move_time`, `turn_time` and `total_time`. 
Using timers

Type in, compile, and run the following program, after predicting what it is meant to do.

task main()
{
    SetSensor(IN_1, SENSOR_TOUCH);  
    ClearTimer(3);  
    OnFwd(OUT_AC, 50);  
    until ((SENSOR_1 == 1) || (Timer(3) > 1000));  
    Off(OUT_AC);  
}
A programming challenge

- We want Hank to be able to:
  - back up, turn right by about 45 degrees, and move forward in case the left sensor only hits an obstacle;
  - back up, turn left by about 45 degrees, and move forward in case the right sensor only hits an obstacle;
  - back up, turn around by 180 degrees, and move in case both sensors hit an obstacle.

- When both sensors hit an obstacle an within a very short period of time, but not exactly at the same instant, we still want Hank to completely turn around.

- Write a program that achieves this task. Hint: You might want to use a timer from the moment a sensor has hit an obstacle to check whether the other sensor will also hit the obstacle within a short time interval.