NXC – Overview

- The NXT has a bytecode interpreter (provided by LEGO), which can be used to execute programs.
- The NXC compiler translates a source program into NXT bytecodes, which can then be executed on the target itself.
- Although NXC is very similar to C, NXC is not a general-purpose programming language - there are many **restrictions** that stem from limitations of the NXT bytecode interpreter.
- The NXC Application Programming Interface (API) describes the system functions, constants, and macros that can be used by programs.
- This API is defined in a special file known as a ”header file” which is, by default, automatically included when compiling a program.

NXC – Main Features

- Multi-Threading support. A task in NXC directly corresponds to an NXT thread

```
task name()
{
// the task's code is placed here
}
```

- A program must always have at least one task - named "main" - which is started whenever the program is run. Maximum number of tasks is 256.
- Scheduling mechanisms

  Example : **Precedes(task1, task2, ..., taskN)**
  - Schedule the specified tasks for execution once the current task has completed executing.
  - The tasks will all execute simultaneously unless other dependencies
- Task priorities

Example

```
mutex moveMutex;
task move_square()
{
  while (true)
  {
    Acquire(moveMutex);
    OnFwd(OUT_AC, 75); Wait(1000);
    OnRev(OUT_C, 75); Wait(500);
    Release(moveMutex);
  }
}
task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
```

```
      Acquire(moveMutex);
      OnRev(OUT_AC, 75); Wait(500);
      OnFwd(OUT_A, 75); Wait(500);
      Release(moveMutex);
    }
  }
}
task main()
{
  Precedes(move_square, check_sensors);
  SetSensorTouch(IN_1);
}
```

Lustre to NXC

- Automatic generation of NXC code from Lustre programs
- Normally, the Lustre compiler produces ansi-C code, too complex to be handled by the nxc compiler. To produce very simple C code which can be compiled by NXC compilers, use Lustre compiler (from version 0.5) with option **-nxc**.
- `lus2c double_counter.lus double_counter -nxc`
  produces a file
  `double_counter.ec2nxc`
  which contains :
  - `void double_counter_I_c(bool)`
    is the input procedure that must be called to feed the program.
  - `void double_counter_step()`
    the procedure that performs one cycle of the program and calls the 2 output procedures :
    `double_counter_O_x(int), double_counter_O_y(int)`
    These procedures should be defined by the user.

Writing a main NXC program

In order to compile and execute the code generated by the Lustre compiler, the user should write a main NXC program that :

1. defines the output procedures,
2. includes the ec2nxc code,
3. defines the main task consisting in a loop that :
   - call the input procedure

     `double_counter_I_c;`

     For a real application the input value should be obtained from the sensors
   - call the step procedure

Example – double counter

```
node double_counter ( c: bool ) returns ( x : int; y : int);
let
   x = (0 -> pre x) + if c then 1 else 0 ;
   y = (0 -> pre y) + if c then 0 else 1 ;
tel
```

Example

```
/* Output procs. <node-name>-O-<var-name>(<var-type>) */
void double_counter_O_x(int V) { NumOut(0, LCD_LINE3, V); }
void double_counter_O_y(int V) { NumOut(0, LCD_LINE4, V); }

/* Includes of the (compiled) Lustre code.
The input proc(s) is(are) defined here, and must be called
at each cycle, before calling the step procedure */
#include "double_counter.ec2nxc"
task main () {
   int cycles_counter = 0;
   bool c = false;
   while (cycles_counter < 3000) {
      //prepares and launches a step...
      cycles_counter++;
      c = !c;
      double_counter_I_c(c);
      double_counter_step(); }
}
```

Periodic Tasks

The rate of the cycles are not related to the ”real-time” : a new cycle begins as soon as the previous cycle ends.

In real-time programming, it is very common that a task should be executed with a known period (e.g. 100 ms). This can be approximated by enforcing the main task to wait between two cycles :

```
task main () {
   int cycles_counter = 0;
   bool c = false;
   while (cycles_counter < 3000) {
      cycles_counter++;
      c = !c;
      double_counter_I_c(c);
      double_counter_step();
Wait(msDelay);
}
```

Problem : It is hard to know the execution time of the step procedure

Periodic Tasks (cont'd)

Modified program : the step call is replaced by a start task statement.

```
task do_one_step () {
   double_counter_step();
}
task main () {
   int cycles_counter = 0;
   bool c = false;
   while (cycles_counter < 3000) {
      cycles_counter++;
      c = !c;
      double_counter_I_c(c);
      StartTask(do_one_step);
Wait(msDelay);
   }
}
```

Delay between two step calls : $msDelay$ + some constant overhead (5 statements).

Periodic Tasks (cont'd)

When the Worst Case Execution Time (WCET) of the step procedure is greater than the expected period, a step will be "re-launched" while the previous step has not yet finished.

We can modify the program in order to check this problem at run time :

```
int nb_problems;
int running;
task do_one_step () {
   running = true;
   double_counter_step();
   running = false;
}
task main () {
   int cycles_counter = 0;
   bool c = false;
   nb_problems = 0;
   running = false;
```

```
  while (cycles_counter < 3000) {
     cycles_counter++;
     c = !c;
     double_counter_I_c(c);
     if(running) nb_problems++;
     StartTask(do_one_step);
     Wait(msDelay);
  }
  TextOut(0, LCD_LINE8, "problems:");
  NumOut(10*6, LCD_LINE8, nb_problems);
  Wait(10000);
}
```

Motor commands in NXC

```
//NXC: TO BE DEFINED BY USER
//ud et ug sont les puissances qui varient de 0 a 100
void Controller_O_u_d(_real ud) {
OnFwd(OUT_A, ud);
}
//NXC: TO BE DEFINED BY USER
void Controller_O_u_g(_real ug) {
OnFwd(OUT_B, ug);
}
```

Configuring and Reading Sensors in NXC

```
SetSensorLight(IN_1, true);
SetSensorLight(IN_2, true);

SetSensorType(IN_1, SENSOR_TYPE_LIGHT_ACTIVE);
SetSensorMode(IN_1, SENSOR_MODE_PERCENT);
SetSensorType(IN_2, SENSOR_TYPE_LIGHT_ACTIVE);
SetSensorMode(IN_2, SENSOR_MODE_PERCENT);

sensD = Sensor(IN_1);
sensG = Sensor(IN_2);
```

- The program first configures port 1 and 2 as light sensors.
- It then configures the mode (scaled value from 0 to 100) and type (with LED on) of the sensors
- It then reads the values of the sensors

Calibration of Sensors

- Note that the controller was designed under the assumption that the value range of a light sensor is $[0, 100]$. In practice, this interval can be different.
- Before executing the control program, it is necessary to determine the real values of the white and black.
- To do so, we write a calibration subroutine and call it before the Main task
  - Point the light sensor on the black zone, the subroutine reads the sensor value and memorizes it as the value for the ”black”
  - Repeat the same procedure for determining the sensor value for the ”white”

Type Definition

Example of a generated ec2nxc program

```
/********
* ec2c version 0.65
* c file generated for node : Controller
* context   method = NXC
* ext call  method = MACROS
********/
/* This program needs external declarations */
#define _Controller_EC2C_SRC_FILE
#define _boolean bool
#define _integer int
#define _false false
#define _true true
/*--------
 * the following ``constants'' must be defined:
extern _real pi;
```

```
extern _real kp_teta;
extern _real ki_teta;
extern _real T;
--------*/
```

$\Rightarrow$ The user needs to define, in the main program, the (generic) type `_real` generated by Lustre. The type `float` of NXC (or by `int`) can be used. Floating point arithmetic will be slower than integer operations !

Type Conversion

- Simulink models allow real number representation in double precision, but NXC allows only float in simple precision (32-bit IEEE 754 single precision floating point) representation.
- Rounding error, a small non-zero number in Simulink can become zero in NXC
  - For the computation that may produce small results, multiply the terms to get a larger number and then scale down in the end to get the true result.
  - Some rules
    1. When adding and substracting, both numbers must have the same scale factor.
    2. When multiplying/dividing, the numbers need not have the same scale factor. The scale factor of the product/quotient is the product/quotient of the scale factors of the original numbers.
- Overfow may occur ! Check the range of floats/int. Some useful constants in NXC :

```
NEG_FLT_MIN   -1E-37
FLT_MIN   1E-37
NEG_FLT_MAX   -1E+37
FLT_MAX   1E+37
```

Summary - Writing a main NXC program for the controller

Must include at least the following :

1. definition of the ”real” type
2. defining the output procedures,
3. including the ec2nxc code,
4. defining the main task :
   - it first calls subroutines to configure sensors, calibrate sensors
   - it then executes a ‘while’ loop
     - call the input procedure
     - call the step procedure

Useful functions for debugging

```
char TextOut  (  int   x,
int   y,
string   str,
unsigned long   options = DRAW_OPT_NORMAL
)
```

Draw a text value on the screen at the specified x and y location.

Example :

```
TextOut(0, LCD_LINE1, "calib blanc", DRAW_OPT_NORMAL);
```

See more in the online manual at

http://bricxcc.sourceforge.net/nbc/nxcdoc/nxcapi/index.html