

Programming Lego/NXT with Lustre

Pascal Raymond

This paper tries to give some hints on how programming the Lego/NXT brick with the data-flow, synchronous language Lustre.

The pdf version is here: [lustre4lego.pdf](#) .

The exemples presented here (and some others) are in this archive: [lustre4lego.test.tgz](#)

1 Requirements

You will need:

- A pc running Linux. The necessary tools will (hopefully) be adapted for macosX and cygwin.
- A recent version of the Lustre/c compiler ($\text{lus2c/ec2c} \geq 0.5$). This tool is included in the full Lustre distribution ($\geq \text{II.k}$):
<http://www-verimag.imag.fr/raymond/tools/lv4-distrib.html>
- A recent version of the nbc/nxc compiler ($\geq 1.0.1.b34$):
<http://bricxcc.sourceforge.net/nbc/>
- The tool `t2n` (talk to next) to transfer, under Linux, executable code to the brick via the USB port:
<http://www-verimag.imag.fr/raymond/edu/nxt/>

2 Principles

2.1 Lustre

Lustre is a data-flow, synchronous language. For an overview on synchronous programming in general, and Lustre in particular, see:

<http://www-verimag.imag.fr/raymond/edu/eng/>

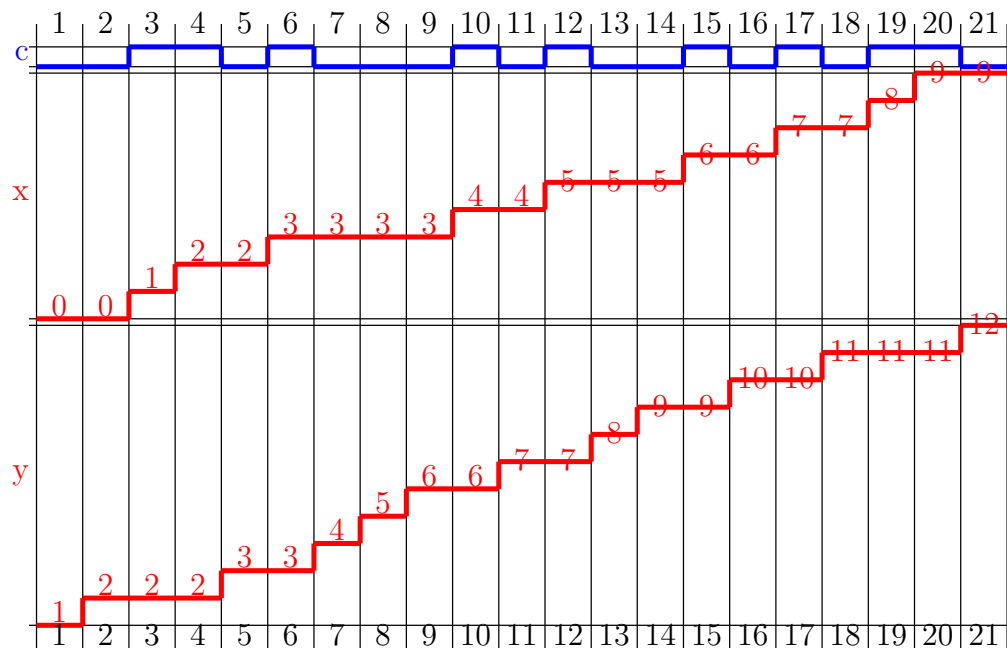
Here is a simple program:

```

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

```

This program has a cyclic behaviour: at each step, depending on the input value c , the outputs x or y are incremented. Here is the timing diagram of an execution:



2.2 Compiling Lustre

Compiling a Lustre program does not produce a proper executable program, but rather an “object” that some main program should animate. For the example it produces:

- a structure containing the internal variables of the object, the user is not suppose to access this “private memory”,
- a procedure performing one cycle of the program (a step), that the user should include into some “main loop”.

Normally, the Lustre compiler produces ansi-C code, too complex to be handled by the nxc compiler. Starting with version 0.5, the lus2c compiler proposes an option `-nxc`, which tells the compiler to produce very simple c code. The command:

```
lus2c double_counter.lus double_counter -nxc
```

produces a file `double_counter.ec2nxc` wich defines:

- `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. This procedure calls the 2 output procedures:
 - `double_counter_O_x(int)`
 - `double_counter_O_y(int)`

Those procedures are not defined: the user must write them.

3 A naive main program in nxc

In order to “run” the example, the user should write a main nxc program that:

- defines the output procedures,
- includes the `ec2nxc` code,
- defines the main task consisting in a loop that:
 - feed the `double_counter` by calling the input procedure `double_counter_I_c`; for a real application the input value should be obtained from the sensors, but here we simply alternate true and false,
 - calls the step procedure,
 - and so on for a while (here 3000 cycles).

```
/*  
A minimal main program for animating  
the double_counter  
*/
```

```
/*  
Lustre output procs. must be defined by the user  
before including the Lustre code.  
The names and profiles can be deduced from the Lustre file:
```

```

    <node-name>-0-<var-name>(<var-type>)
    The implementation can be whatever you want...
    (in this example, outputs are printed on the screen)
*/

void double_counter_0_x(int V) { NumOut(0, LCD_LINE3, V); }
void double_counter_0_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();
    }
}

```

4 Implementing a periodic task

4.1 A wrong solution

In the previous main program 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);
    }
}

```

The problem is now to define `msDelay` in such a way that two consecutive calls of `double_counter_step` are separated by the expected period. In fact, this is almost impossible since it depends on the execution time of the step procedure (which is unknown, and, moreover, is certainly not a constant).

4.2 A better solution

In order to approximate more accurately a periodic behavior, we introduce a new task which is responsible of calling the step function. In the main loop, 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);
    }
}

```

With this new version, we can have a precise idea of the delay between two step calls: `msDelay` + some constant overhead (5 simple statements). In a first approximation we can consider that the overhead is neglectable, and set `msDelay` to the expected period (e.g. 100 ms). If this approximation is not good enough, we can set it to 99 ms, 98 ms etc.

4.3 Perfecting the solution

The previous solution can be considered as satisfactory as far as the *Worst Case Execution Time* (WCET) of `double_counter_step` is less than the expected period. If it is not the case, 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);
}
```

Note that the variable `running` is shared between the two tasks. In “clean” parallel programming, access to shared resources should be made exclusive by using some “mutex” mechanism. But in this precise case it is not necessary: the NXP abstract machines guarantees that a single assignment is always performed as an “atomic” piece of code.