SimPyLC HowTo

Author: Jacques de Hooge

"Simplicity is the hallmark of the true"

Table of contents

QQuickLicence governing the use of SimPyLC and of its generated code

This license governs use of the accompanying software ("Software"), and your use of the Software constitutes acceptance of this license.

You may use the Software for any commercial or noncommercial purpose, including distributing derivative works.

In return, it is required that you agree:

1. Not to remove any copyright or other notices from the Software.
2. That if you distribute the Software in source code form you do so only under this license (i.e. you must include a complete copy of this license with your distribution in a plain text file named QQuickLicence.txt), and if you distribute the Software solely in object form you only do so under a license that complies with this license.
3. That the Software comes "as is", with no warranties. None whatsoever. This means no express, implied or statutory warranty, including without limitation, warranties of merchantability or fitness for a particular purpose or any warranty of title or non-infringement. Also, you must pass this disclaimer on whenever you distribute the Software or derivative works.
4. That neither Geatec Engineering nor any contributor to the Software will be liable for any of those types of damages known as indirect, special, consequential, or incidental related to the Software or this license, to the maximum extent the law permits, no matter what legal theory it's based on. Also, you must pass this limitation of liability on whenever you distribute the Software or derivative works.
5. That you will not use or cause usage of the Software in safety-critical situations under any circumstances.
6. That if you sue anyone over patents that you think may apply to the Software for a person's use of the Software, your license to the Software ends automatically.
7. That your rights under this License end automatically if you breach it in any way.
8. That all rights not expressly granted to you in this license are reserved.

Introduction

This document briefly describes the functions of the SimPyLC control simulator. Purpose of the simulator is to allow students to get acquainted with the design of real time controls for automated production and transportation systems. The controlled systems may range from out of the box assembly line robots to special purpose robot cranes and automatically controlled transportation vehicles.

Real time control of such systems often utilize a PLC (Programmable Logic Controller). A PLC has to perform a lot of tasks in parallel. It does so using design patterns that are very different from the ones used in mainstream design of non-control programs. When it comes to performing tasks in parallel, traditionally things like interprocess communication, threads, semaphores, queues en critical sections may come to mind.

Synchronizing tasks by means of these facilities is error prone. Problems may range from deadlocks, where processes are waiting for each other forever, to race conditions, where it's unclear which action will be performed first. Use of queues as means of interprocess communication opens the possibility of control commands being accumulated unwantedly, taking effect at an unexpected moment. Suddenly the system (which may be as small as an electronics welding robot, but also as big as a container crane) starts executing a movement, which may be very dangerous.

Safety plays a major role in designing real time control systems. Even a small production robot may cause harm in an environment where also human labor is being performed. The main demand to be made on a real control system is that the environment is being watched constantly and that actions are not solely dictated by supervisory commands and previous program state, but also by constant evaluation of sensor input from the environment.

To be able to experiment with controlling real hardware, a code generator for the well known Arduino processor board has been added. **Note that the simulator as well as the code generator may contain bugs. So the generated code is only suitable for educational purposes and controlled systems should be chosen not to pose any risk of damage or injury in case of control malfunction.** A control can be simulated in advance and when it is ready, C code can be generated and uploaded to the Arduino.

Installation

Windows:

1. Install Python 3.8 from https://www.python.org/downloads/
2. Open an command prompt
3. Type *python -m pip install simpylc*
4. Fetch Windows accessories into current directory by typing *python -m simpylc -a*
5. From the current directory, copy *freeglut64.vc14.dll* to *C:\Windows\System32*
6. From the current directory, optionally install the *QuartzMS.TTF* font as desribed in the Windows docs

The latter two steps may require administrator privilege. If you use Windows Explorer, you should be prompted for that.

Linux:

1. Install MiniConda for Python 3.8 from https://conda.io/miniconda.html
2. Open a command prompt
3. Type *conda install numpy*
4. Type *conda install pyopengl*
5. Type *python -m pip install simpylc*
6. Perform one of the following alternatives:
    1. Install FreeGlut as explained on http://freeglut.sourceforge.net/index.php#download.
    2. Type the following command sequence:
    sudo apt-get update
    sudo apt-get install build-essential
    sudo apt-get install freeglut3-dev

OSX:

1. Install MiniConda for Python 3.8 from https://conda.io/miniconda.html
2. Open a command prompt
3. Type *conda install numpy*
4. Type conda install pyopengl
5. Type *python -m pip install simpylc*
6. Perform one of the following alternatives:
    1. Install FreeGlut as explained on http://freeglut.sourceforge.net/index.php#download
    2. Type *brew install freeglut*
7. Install the XQuartz X-window system from https://www.xquartz.org
8. If during use the message No matching pixelformats found is displayed then type *export LIBGL_ALLOW_SOFTWARE=1* prior to starting SimPyLC, or adapt this environment variable permanently.

For all OS'es, type *python -m simpylc -h* to learn how to view the docs and fetch the example simulations.

GUI quick reference

- [LEFT CLICK] on a field or [ENTER] gets you into edit mode.
- [LEFT CLICK] or [ENTER] again gets you out of edit mode and into forced mode, values colored orange are frozen.
- [RIGHT CLICK] or [ESC] gets you into released mode, values are thawed again.
- [PGUP] and [PGDN] change the currently viewed control page.
- [WHEEL PRESS] on a marker field makes it 1, release makes it 0 again, both without freezing it.
- [WHEEL ROTATE] changes the value of a register field, without freezing it.

Putting together a simulation

Two example simulations are part of the SimPyLC distribution.

The one to start with is in the *blinkingLight* directory. To understand what is happening, run the example in that directory from the command line: *python world.py*. Look at the sourcecode in *blinkingLight.py* and at the sequence diagram of the running application to see causes and effects. Also take a look at *timing.py* to see how the sequence diagram is defined and at *world.py* to see how the parts of the simulation are tied together.

The second example is in the *oneArmedRobot* directory, and consists of the following files:

world.py	Defines the parts that make up the simulation
robot.py	A SimPyLC module that simulates the physics of a production robot
control.py	A SimPyLC module that contains the control software for the robot
visualisation.py	Contains an OpenGL visualisation of the robot
timing.py	Draws sequence diagrams of selected signals

Start the example simulation from the command line in the same way as the first example: *python world.py*. Play around with it. A control element gets in the input state by a [left click] or by pressing *[enter].* Then you can input a value. *[Left click]*ing or pressing *[enter]* again will force the control element to retain the value you specified. *[Right click]*ing or pressing *[esc]* will release the control element, allowing it to be overwritten.

Start by reading through *robot.py* using the circuit descriptions above, and try to find out what makes it tick. Then shift your attention to *control.py* for a little more complexity. Pay special attention to the **input** methods of both the **Robot** and **Control** class, to gain insight in how these modules (both inherit from class **Module**) exchange control signals.

Then take a brief look to into *visualisation.py* to find out how a so called hierarchical model works. Peek into the file *graphics.py* to see how OpenGL is involved. Google for "OpenGL Hierarchical Modeling" to find many explanations of how this works.

Also take a look at *timing.py* to see how the channels of the sequence diagram are configured.

Basic control elements

```
class Marker:
        # Boolean expression evaluator
        def __init__ (self, condition = False):
                # Sets the boolean value of the marker to  condition
        def mark (self, trueValue, condition = True, falseValue = None):
                # Sets the boolean value of the marker to trueValue if
                condition is True
                # Sets the boolean value of the marker to falseValue if:
                #       - condition is False
                #       and
                #       - parameter falseValue is present and not None
                # Leaves the boolean value of the marker unaltered if:
                #       - condition is False
                #       and
                #       - parameter falseValue is absent or None

class Oneshot:
        # Edge detector
        def __init__ (self, condition = False):
                # Initializes the boolean value of the oneshot to condition
        def trigger (self, condition):
                # Sets the boolean value of the oneshot to True if:
                #       - condition is True
                #       and
                #       - condition was False previously
                # Resets theboolean value of the oneshot to False in all
                other cases
                # So a oneshot can only remain True for at most one sweep

class Latch:
        # Boolean memory cell
        def __init__ (self, condition = False):
                # Sets the boolean value of the latch to condition
        def latch (self, condition):
                # Sets the boolean value of the latch to True if condition is
                True
                # Leaves it unaltered if condition is False
        def unlatch (self, condition):
                # Sets the boolean value of the latch to  False if condition is
                True
                # Leaves it unaltered if condition is False
```

```
class Register :
        # Numerical expression evaluator
        def __init__ (self, value = 0):
                # Sets the numerical value of the register to value
        def set (self, trueValue, condition = True, falseValue = None):
                # Sets the numerical value of the register to trueValue if
        condition is True
                # Sets the numerical value of the register to falseValue if:
                #       - condition is False
                #       and
                #       - parameter falseValue is present and not None
                # Leaves the numerical value of the register unaltered if:
                #       - condition is False
                #       and
                #       - parameter falseValue is absent or None

class Timer:
        # Total elapsed timer
        def __init__ (self):
                # Resets the seconds value of timer to 0
        def reset (self, condition):
                # Resets the seconds value of timer to 0 if condition is True

class Runner:
        # Special singleton Marker allows freezing a simulation  by halting
        the clock
        def __init__ (self, condition = True):
                # Sets the boolean value of the runner to True
```

The PLC programming paradigm

PLC software, whether running on special hardware or on a PC, adheres to some very simple rules that seem overly strict at first sight. However in practice it has turned out that following these rules lead to stable, predictable system behavior. So for one moment let's forget the richness of modern programming languages and operating systems and delve into a very different world: The world of real time control software.

Central in each PLC program is a loop that is executed forever. In a general purpose programming language the code is something like:

```
while True:
    readInputFromSensorsAndControls ()      # Controls can be switches, commands etc.
    calculateOutputFromInputAndPreviousState ()
    writeOutputToActuatorsAndIndicators ()  # Indic. can be lamps, meters, status reports etc.
```

PLC's have their own terminology. Going through the above loop once is called a **sweep**. The time it takes to perform one sweep is called the sweeptime. In the simulator, the sweeptime is available in the variable **world.period**. The sweeptime determines the reaction time of the PLC. It may range from a few to a few dozen milliseconds.

- Within one sweep, at first all input from sensors, operator controls and external control systems is read. This is indicated by the function **readInputFromSensorsAndControls ()**.
- Next all output signals are computed from the inputs and from the previous control state. This is indicated by the function **calculateOutputFromInputAndPreviousState ()**.
- Finally, in the function **writeOutputToActuatorsAndIndicators ()**, the outputs are transferred to the outside world, e.g. servo motors, electromagnets, indicator lamps to inform humans or signals and reports for external control systems.

After this, the whole sequence starts all over again: the next sweep. In a real world PLC, the sweeptime is guarded by a watch dog timer. If it exceeds a maximum, the system is brought to a halt in a safe manner.

What is crucial to the way PLC programs operate, is that all inputs are read at the start of every sweep. So what e.g. is bad style for a PLC program, is to have a sensor report transient information. For instance as long as the robot does not hit limit switch s1, this leads to a **continuous** input **s1Enabled** rather than having a brief spike **s1Hit** at the moment the robot hits the limit switch. The advantage is

that is case of sensor malfunction or software glitches, the system will come to a halt, rather than run out of control.

To get a better feeling for the benefits of this approach, let's look for a moment at the oil level indicator light in a car. The worst design is a lamp that only briefly lights up if the car runs out of oil. The driver will almost certainly miss this transient information. A little better is a lamp that will remain burning as long as the oil level remains too low. From the standpoint of system failure however, a green lamp that burns as long as the oil level is ok, is preferable, for if the lamp itself fails, the driver will halt the car, thereby avoiding the risk of damaging the engine by lack of lubrication. Since a lamp switching off draws less attention than a lamp switching on, a solution where a green light indicates enough oil and a red or blinking light or no light at all (lamp failure) indicates a problem, is probably the optimum.

PLC programming rules

The first step in programming safe controls is to follow a couple of rules. As said, these rules seem very strict, even unworkable and draconic at first. But let firsthand experience be your teacher here. Follow them consequently in your designs and it will quite soon become clear what the benefits are. Here are the rules:

1. No loops other than the main sweep. So no **while**, **for**, **repeat**, **goto** etc.
2. No conditional statements. So no **if**, **elif**, **else**, **switch**, **case** etc.
3. No pause, delay, sleep or anything like that
4. Be brief and concise

In short: No control structures at all!

How can anyone make any real world computer program like this? Surely control structures have not been a part of every mainstream language right from the advent of computing for no purpose... All that you can do without them is evaluate and assign expressions. How to control anything by doing only that?

Well, the answer is that there certainly is a category of languages that lacks imperative control structures, in fact it lacks imperative statements at all. This category is the category of **functional languages**. Functional languages boil down to stating functional dependencies between input and output. Functional languages are generally considered to hold great promise when it comes to system reliability. But they are also considered to be strange beasts, only to be comprehended by mathematicians that talk to their wives in formulae.

It is the functional programming paradigm that PLC controls borrow their reliability from. The most reliable PLC control is a PLC control where the output of one sweep is a mere function of its instantaneous input, not relying on any state from previous sweeps. This statelessness is what makes functional programs reliable and easily testable.

In practice, PLC programs do retain state from previous sweeps in the form of registers, latches and oneshots, to be discussed later. However, the less state is retained, the more reliable the control, even when this means that some seemingly redundant computations are done over and over again in every sweep.

The golden bullet of designing reliable control programs is to know when to retain state and when not. This is a matter of common sense and experience. The former boils down to a choice: get acquainted with the textbook way of programming in any 3rd generation language (C, C++, Python, C#, Java, JavaScript, Fortran, Basic, Pascal, Modula, Ada) using all the bells and whistles that these ever growing languages offer. After that, imagine yourself standing next to a 40 ton coal grab unloader, swaying at full speed at four meter distance from your head, controlled by a computer. And ask yourself: how much program

complexity do I dare to trust my life to. The latter, experience, is what you hopefully will gain by building robot controls using the simulator.

The third point of the PLC programming rules, be brief and concise, is probably the most important one. One way to obtain reliable software is intervision. Such intervision, called deskchecking or peer reviewing if software is involved, relies on the observation that explaining something to another person is a very good way to reach a thorough understanding of it yourself. You and a colleague go through the program step by step. Explain to your colleague what you have done and why, and you'll discover your own mistakes. And maybe your colleague will discover some too, but that's just a bonus.

To be able to go through a program in this way, it must not be too bulky. Reading through ten pages of sourcecode and keeping track of the interactions is doable. Reading through twohundred pages will make you lose track. Encapsulation techniques like object oriented programming are one way to tackle complexity. Another way is simplicity. PLC programs tend to be simple, and, strange enough, they tend to be brief. This is counterintuitive. Certainly being restricted to a small subset of program structures will lead to lengthy sourcecode? In practice this turns out not to be the case. Why this is so is the subject of the next two paragraphs.

What is wrong with encapsulation and control structures?

What is wrong with encapsulation and control structures? Well, nothing of course, encapsulation is hiding local and volatile design decisions behind a well defined stable interface. It is the cornerstone of object oriented programming and is quite successful if applied at the right occasions.

PLC's use encapsulation intensively. The "electrical circuit"-like elements that make up a PLC program (timers, registers, latches, oneshots etc.) are all instances of classes, hiding their internal complexity behind a very simple and well defined interface.

But this is system software, which means: it is thoroughly tested part of the PLC itself, not of the ever changing control programs you write for it. So encapsulation does its work behind the scenes.

Something similar holds for control structures like loops (**while**, **for**, **repeat**, **goto**) and conditional statements (**if**, **elif**, **else**, **switch**, **case**). The whole PLC concept revolves around one central loop, the sweep. And to facilitate efficient IO, even the dreaded multithreading and interprocess communication are utilized by the PLC itself. And of course the system software that is a fixed, non user programmable part of any PLC, uses conditional statements. But as said: this complexity is restricted to exhaustively tested system software in any real PLC (as opposed to a simulator for learning purposes).

The control program running on top of this system software should be simple, straight and brief. Fortunately, switching from a typical object oriented programming style to the straight and simple style that is the norm for real time controls, lessens the amount of sourcecode. Program size tends to shrink from hundreds of pages of sourcecode to tens of pages. The fact that this is so, is not caused by any flaw of object oriented programming, but by the rather unusual characteristics of robot controls, as explained in the next paragraph.

But what makes controlling a robot system so different?

Good question! Real time control programs **are** different from mainstream programming. In a nutshell, here's why. A robot control generally has the following properties:

1. The amount of sensor signals (inputs from the outside world) typically is four times the amount of actuator signals (outputs to motors, magnets, welding electrodes, paint sprayers etc.). So a control system e.g. needs input from forty sensors to control only ten actuators.
2. To determine any of the outputs, typically almost all sensor signals are needed, either directly or indirectly. Look at yourself as a robot. For even a routine task like determining when to put the fire low when roasting meat, you'll use sight, hearing, smell and tactile input. For automated control systems it turns out to be the same in practice.

Let's see what this means for noble concepts like encapsulation on the application (rather than internal PLC system) level. Since every output depends upon tens or hundreds of inputs, the public interface of classes in object oriented controls (or the parameter lists in function structured controls) turns out to be excessively large.

Another observation is that while control program parts for e.g. the upper arm, the forearm and the wrist of a robot control at first sight look similar, in practice they differ in minute but essential ways. So at best they can be modeled by related classes using inheritance. But the differences are ad hoc and may change completely during the productive life of the control system, if e.g. safety switches are added or a new sequence of robot acts becomes necessary to manufacture an altered or new product. There's a tendency for changes like that to completely wreck even the most carefully designed inheritance hierarchy.

The only weapon here is simplicity. Work straight towards the task at hand without additional restrictions like stable interfaces, since they will make the program grow manifold, to be prepared for an unknown future.

At this point some relativation and reservation is needed. Program design is striking the right compromises. All the rules so forcefully advocated above have exceptions. Pragmatism is to be preferred over dogmatism, especially with control systems. And there are control systems that do not fit well with the rules stated above. Keep reading.

Control levels

In order to determine what approach will work best, it's needed to be aware of the following control hierarchy:

4	Logistic control level	Planning sequences to achieve a final goal: Load all the boxes from the stockpile on the left upon the palette on the right.
3	Sequence control level	Chaining atomary commands to achieve an intermediate goal. E.g. displacement: Grab that box and put it overthere.
2	Direct control level	Execute atomary commands like: Grab that box. Or: Put it over there.
1	Hardwired locking level	Maintain safety by means of hard wired negative (often tactile) sensors. No signal means: Stop .
0	Inherent safety level	Physical prevention of accidents: Arm not long enough to reach operator. Concrete buffer at the end of the track.

It's the levels 2 and 3 that the PLC programming style treated in the previous paragraphs is most suitable for. Level 4 is better tackled using mainstream object- or data-oriented programming. Level 1 is the domain of special purpose hard wired electrical circuits and a limited amount of simple and robust sensors. Level 0 is the level where the quay turns the ship.

Apart from the above hierarchy, there's a tendency for functionality to move from the control itself into the sensors and actuators. High speed feedback loops are part of the servo motors driving the robot's movements, rather than of the central program. The same holds e.g. for numerical integration of accelerometer data to obtain speed and position. As far as these intelligent actuators and sensors are robust, the trend towards decentralization is beneficial, since it replaces expensive and vulnerable special purpose software by out-of-the-box components. However it may be desirable for the central control to keep a watchful eye on these intelligent components, since the central control is the one able to combine all input information and draw conclusions, e.g. about sensor failure.

Keeping the possibility of sensor failure in mind, it may seem like a good idea to have lots of redundancy in the sensorical capabilities of the robot, but there's a downside to this. If sensors differ of opinion, what should the control do? Halt the system? But that often means halting production. The more sensors are present, the more sensor failure will occur. In some situations a minor chance on limited damage is to be preferred over a robot that regularly blocks a whole production line for no reason other than failure of a redundant sensor.

Putting a robot system into active operation

Imagine a production line of a car manufacturer. The keyword here is: uptime. Every unproductive quarter costs a car worth of money. So you carefully design your control program and, even better, you test it in advance by means of simulation, typically reducing the time to get the robot up and running from weeks to hours. But still.

Real world circumstances differ from the information (requirements) you got in advance. The adjustment of the conveyor belt is more inexact than was specified. Reflections and direct sunlight misguide your optical sensors. Temperature changes of the environments lead to needless emergency stops, as the control is fooled into thinking that the welding transformer is overheated. And so on. It shouldn't be so, but it is. Always.

There are two ways of dealing with this. One way is hiding behind hundred page contracts, drawn up by your companies law department. Any differences from specification? Tough luck for the customer. Claims won't apply, it's all excluded by the legal guys. As an unfortunate side effect however, there will be no more assignments from this customer. Tough luck for your company. Because that's what industrial control programming practice is like. Either live in the real, physical, unpredictable, ever changing world in which your robot control has to live, or don't.

Fortunately most companies are aware of the fact that customers are not merely a nuisance. So their practical needs will have to be met. And those needs include ad hoc change and minimal disruption of production.

What this means for robot control programs is the need for efficient debugging and alteration. You will have to be able to experiment and repair on the spot, under the real circumstances, interacting with real hardware and other automated systems.

To this end, every PLC control allows for real time inspection and alteration of all control objects (timers, latches, oneshots, registers etc.). Arbitrary values can be input during active operation, all values can constantly be monitored by the operator and, last but not least, values can be forced permanently. This means that rather than acquiring an input signal value from a sensor, or computing an intermediate value or outputing a signal value to an actuator, these signals are coerced to have a certain value. This makes rapid, real time, testing and debugging feasible.

That's what all the blinking lights and fluctuating numbers on a control rack are about, not about trying to look like StarTrek, but to both know what's going on and be able to influence it in real time.

The SimPyLC simulator

There are many PLC simulators available for free. Most of them good looking and feature rich. Why another simulator? The Sim in SimPyLC stands for simulation, but it also stands for simple. SimPyLC is a simplified version of its C++ counterpart SimPLC, written by the same author, that has been in use in the industry controlling installations ranging from harbor cranes to chemical production facilities for over twenty years now. SimPyLC is completely written in Python. What this means is that there's so little sourcecode, that you can read through every line of it, if you want.

You can dissect it, alter it to your needs and taste and understand every single line of code. The amount of insight this will give cannot be matched be any theoretical explanation of what such code should do. At first try to write some simple controls using SimPyLC. Use the example code (one-armed production robot) as tutorial material.

Then go on with the more complex assignments. And, if there's a need or an opportunity, take a peek at the sourcecode of SimPyLC itself. So you'll know that there's no magic involved. The goal is to gain the insight and confidence that you could have written it yourself and that you understand exactly how it behaves and why. This is how PLC's work. And hopefully the assignments make clear why these simple and robust control devices are ubiquitous in the world of industrial production and logistics.

Example of an assignment linked to elementary physics

As a starting point in learning to program real time controls, after you understand the code of all simulation examples, you may try to optimize the *oneArmedRobot* control. Especially the torso movement has a lot of overshoot. This is because it is programmed without paying any attention to the underlying physics.

In general, movements are governed by the following equations:

```
s(t) = s(0) + v(0) t + 1/2 a(0) t^2          [1]
v(t) = v(0) + a(0) t                         [2]

v(t) = 0                                     [3]

[2] & [3]      0 ==> v(0) + a(0) t           [4]
[4]    ==>     t = - v(0) / a(0)             [5]

s(t) = 0                                     [6]

[1] & [5] & [6]==>
0 = s(0) + v(0)(- v(0) / a(0) + 1/2 a(0) (v(0)^2 / a(0)^2)
0 = s(0) - v(0)^2 / a(0) + 1/2 v(0)^2 / a(0)
0 = s(0) + (- 1 + 1/2) v(0)^2 / a(0)
0 = s(0) - 1/2 v(0)^2 / a(0)
s(0) = 1/2 v(0)^2 / a(0)
```

Where:

- t is time
- s is position or angle
- v is velocity or angular velocity
- a is acceleration or angular acceleration

`s(0)` denotes the braking distance. A fast control will:

- Accelerate towards the target maximally if further away from the target than the braking distance.
- Accelerate away from the target (decelerate towards the target) maximally if closer to the target than the braking distance

Try to implement such a bang-bang control in the *oneArmedRobot* example. Although this boils down to changing just a few lines of code, it is not as easy as it may seem. You'll have to pay good attention to direction. If you've done this properly, you'll find that no visible overshoot remains and positioning becomes very fast. This is the way e.g. gantry cranes are positioned. The bang-bang effect is damped by the power electronics and inertness of the mechanics in practice if the PLC sweeptime is short enough.

Code generation for the Arduino

To enable controlling real hardware, C code generation for the Arduino processor board has been added. This is a recent feature and has not yet been tested thoroughly. Still, in an educational situation, it is an interesting possibility. Code can be designed and tested using the simulator. Once it's functioning well, it can be uploaded to the Arduino.

Apart from the code generated from your simulation modules, you'll need some I/O code, as can be seen in the file *native.cpp* that is part of e.g. the *arduinoLed* example. As can be seen there, it contains the basic C code that is part of every sketch, along with I/O definition and handling. Also in the *loop ()* function, between input and output, a call to the generated function *cycle ()* must be made. Anothor example of such a *native.cpp* file can be found in the *arduinoTrafficLights* example.

After having done all this, add the modulenames for which one wants to generate code to the command line.

python world.py robot

If you want to generate code for all modules, use:

*python world.py **

To avoid name clashes a prefix can be added to all generated identifiers, e.g.

*python world.py * prefix=PLC*

will prefix all generated identifiers with *PLC_*. Note that in the case of generating code for multiple modules, module prefixes will be inserted as well. These should also be present in *native.cpp*. Don't be intimidated by the seemingly complex prefix stuff. In most cases generating code for only one module and using no prefixes at all will do fine. Just look at the generated code if you need to know how prefixing works.

In all cases the name of the generated *.ino* file will be *<simulationDirectoryName>.ino*. To run the generated code on the Arduino, load this file into the Arduino IDE and then upload it to your board.

The arduinoLed example

The *arduinoLed* example simulation operates the onboard Arduino light emitting diode. The *native.cpp* file in this case has the following contents:

```
void setup () {
        pinMode  (13, OUTPUT);
}

void loop () {
        cycle ();
        digitalWrite  (13, led);
}
```

The arduinoTraficLight example

The *arduinoTrafficLight* example requires a bit more hardware, but not too much. The subject is a four legged crossing with a red and a green light at each leg. The hardware used was an Arduino Due, but should also run on an Arduino Uno if PWM is used and the port addresses are adapted in *native.cpp.*

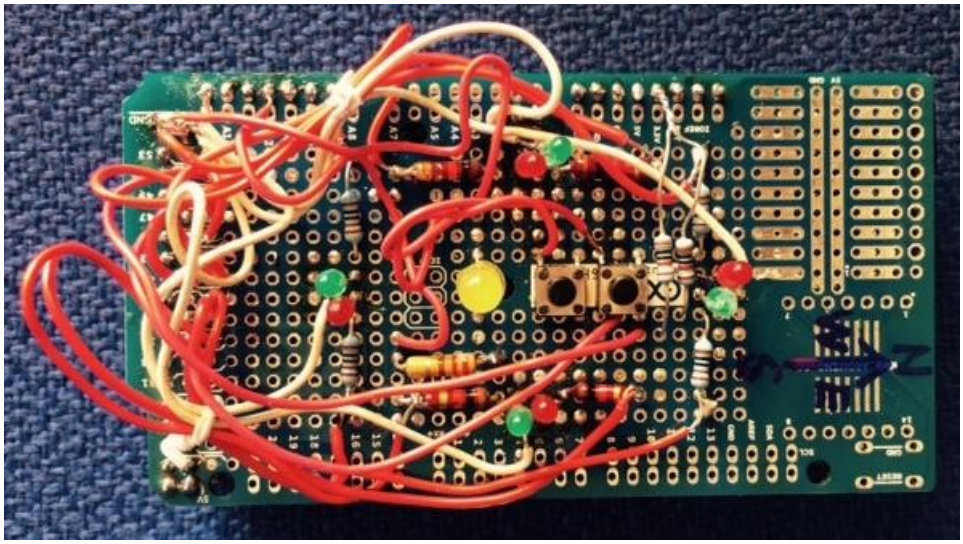

Figure 1. Hardware used to test the traffic light example.

The traffic lights have four modes of operation:

1. *regular* - The north and south leg lights show the same color, as do the east and west legs. If the lights for a certain leg pair has been green for some time, they will start to blink to signal that they will turn red before long.

2. *cycle* - One leg at a time gets green light, rotating clockwise. Before turning red, the green light will blink for a while.

3. *night* - All red lights blink, all green lights are off.

4. *off* - All lights are switched of permanently.

Switching between modes happens by pressing a pushbutton. As an extra there's a central lamp post (big yellow led) that can be dimmed or brightened by pressing and holding another pushbutton. Each time the button is released a switch is made between dimming and brightening.

Writing non-trivial controls like the *oneArmedRobot* using SimPyLC is **much** simpler than programming the Arduino in C directly. The test facilities of SimPyLC

contribute a lot to this, as does the presence of circuit element types like Oneshot and Timer.

After e.g. writing and testing the traffic light control in the simulator, it ran flawlessly in the Arduino at first attempt. This is typical for the use of simulation, also for installations much more complicated than a traffic light control. In fact the author of this document has met cases where after months of commissioning, a control was completely rewritten and debugged in the simulator without even a look at the existing code, tested in a day and commissioned in one hour and a half, no modification needed. A control is either open to understanding or it isn't. In the latter case throwing it away and starting with a fresh slate will save time and accidents.

Take your time to really grab the PLC coding style as exemplified in *oneArmedRobot*, using simulation and timing charts to debug your control even before the hardware is available and you'll soon be writing complex controls with a minimum of effort.

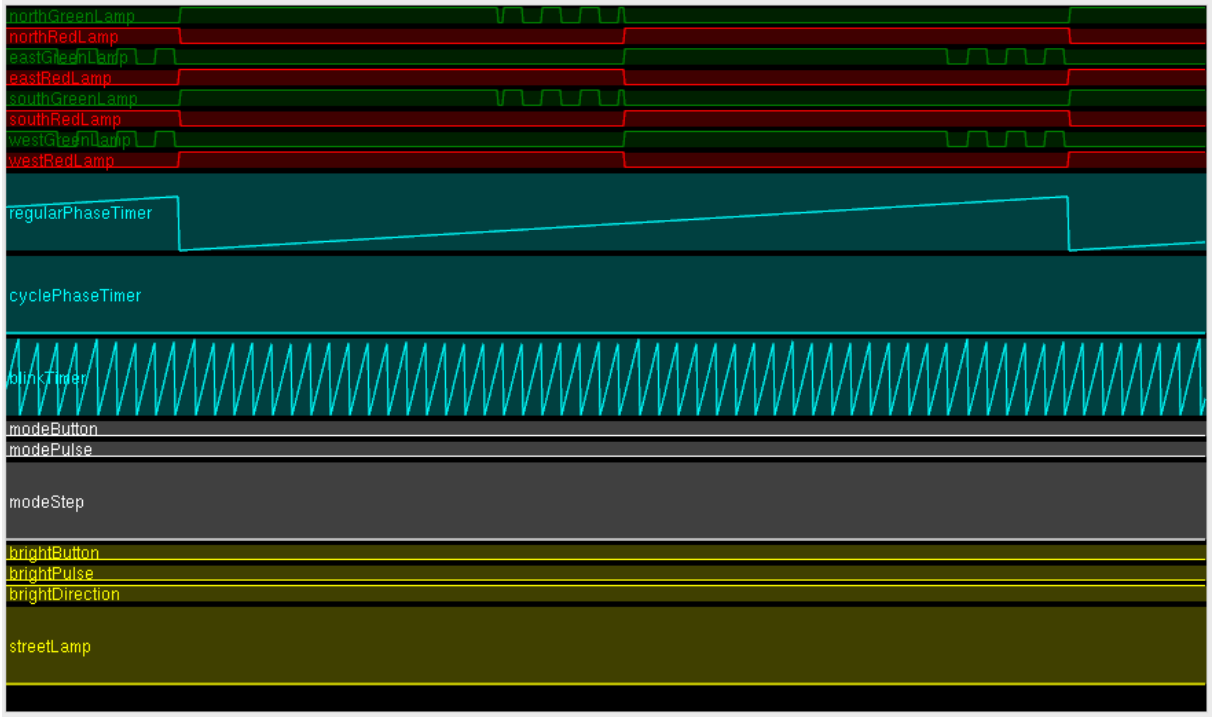

Figure 2. Timing chart of the traffic light example in regular mode.

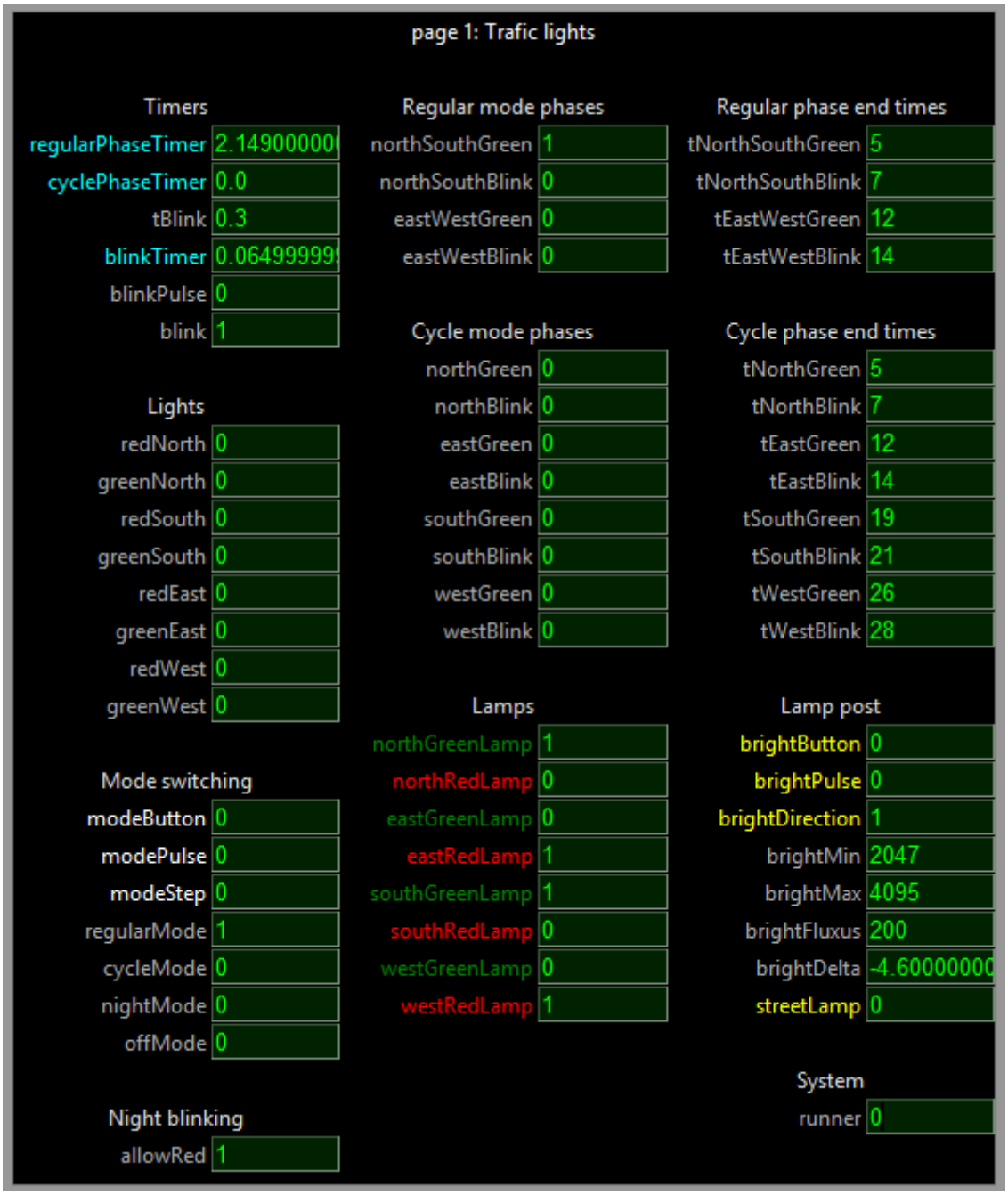

Figure 3. Control panel of the traffic light example.

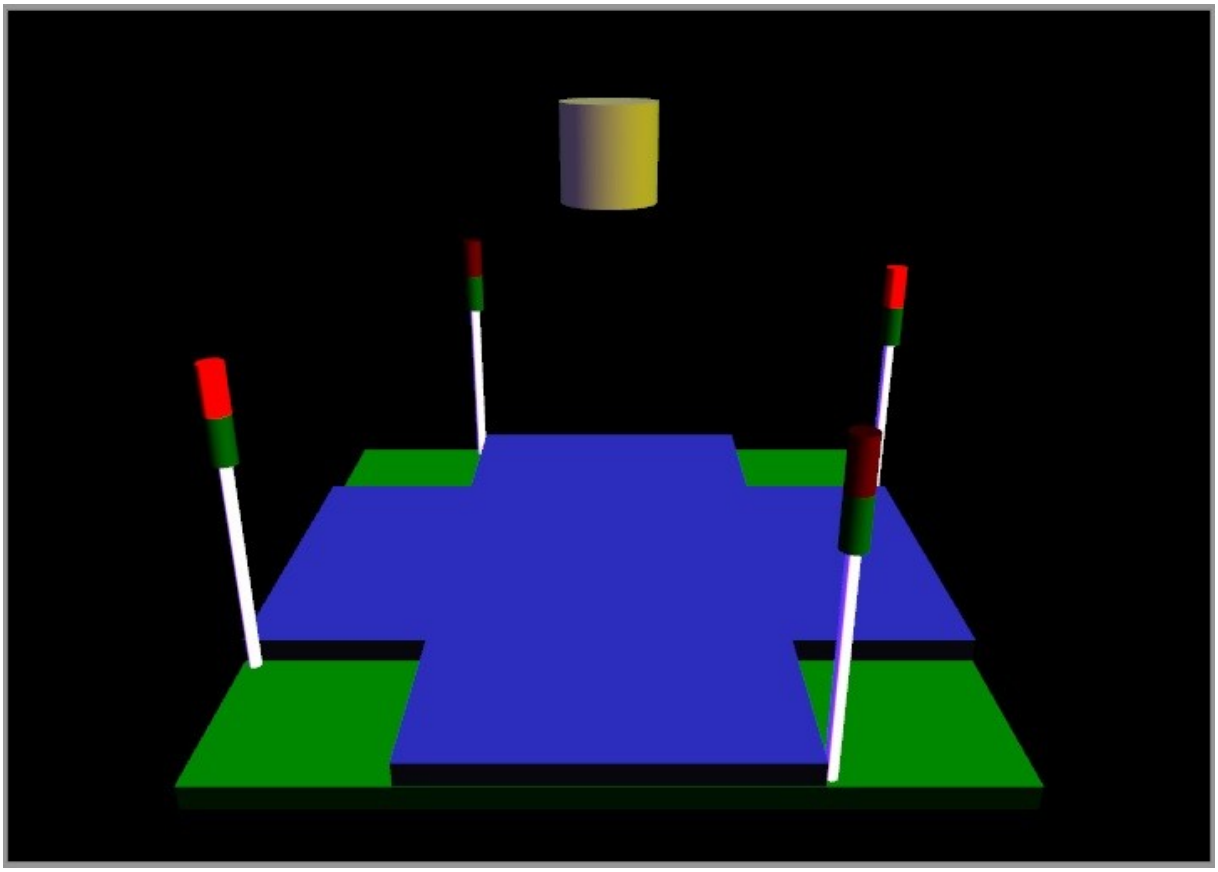

Figure 4. Visualisation of the traffic light example.

The arduinoStove example

The *arduinoStove* example mimics an electric stove with four boiling plates, a cooking alarm clock with buzzer and a child lock. A four digit numerical display is used to indicate the temperature of the plates, the time remaining before alarm and the status of the child lock. Also for each plate a LED is present, its brightness increasing with temperature.

Once power is switched on, use the plate select button to select a plate and then the up and down buttons to adjust its temperature. The dot of the digit of the selected plate lights up. Pressing the child lock button for more than five seconds will lock c.q. unlock the child lock. If the child lock is active, pressing any other button will generate an alarm, resulting in a buzzer sounding with varying pitch. Pressing the alarm select button allows for adjustment of the alarm time with the up and down buttons. The longer these buttons are pressed, the more speedy the adjustment changes.

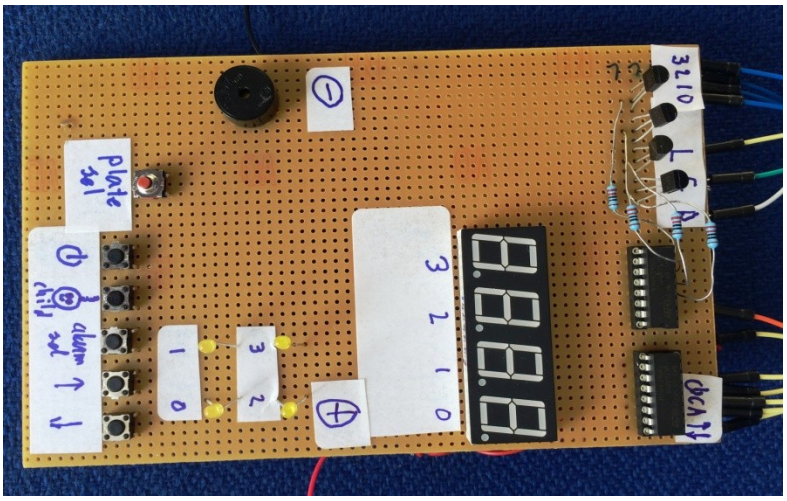

Figure 5. Hardware used to test the stove example.

A visualisation has been added. To make the simulation more interesting, the four digit display is multiplexed. Since its multiplexing frequency interferes with the window refresh rate, it will blink irregularly in the simulator. The buzzer is visualized by a flashlight. To properly view its flashes in the simulation, lower its *buzzerBaseFreq* to e.g. 4 Hz.

Figure 6. Visualisation of the stove example. Only one digit lights up in this snapshot, since the digits are multiplexed.

To make the whole thing work with an Arduino One, you'll need two shift registers and a couple of transistors. Since this example requires quite some hardware and soldering, it is not the most obvious example to start with. It has been added because it resembles an assignment that my students have to work on.

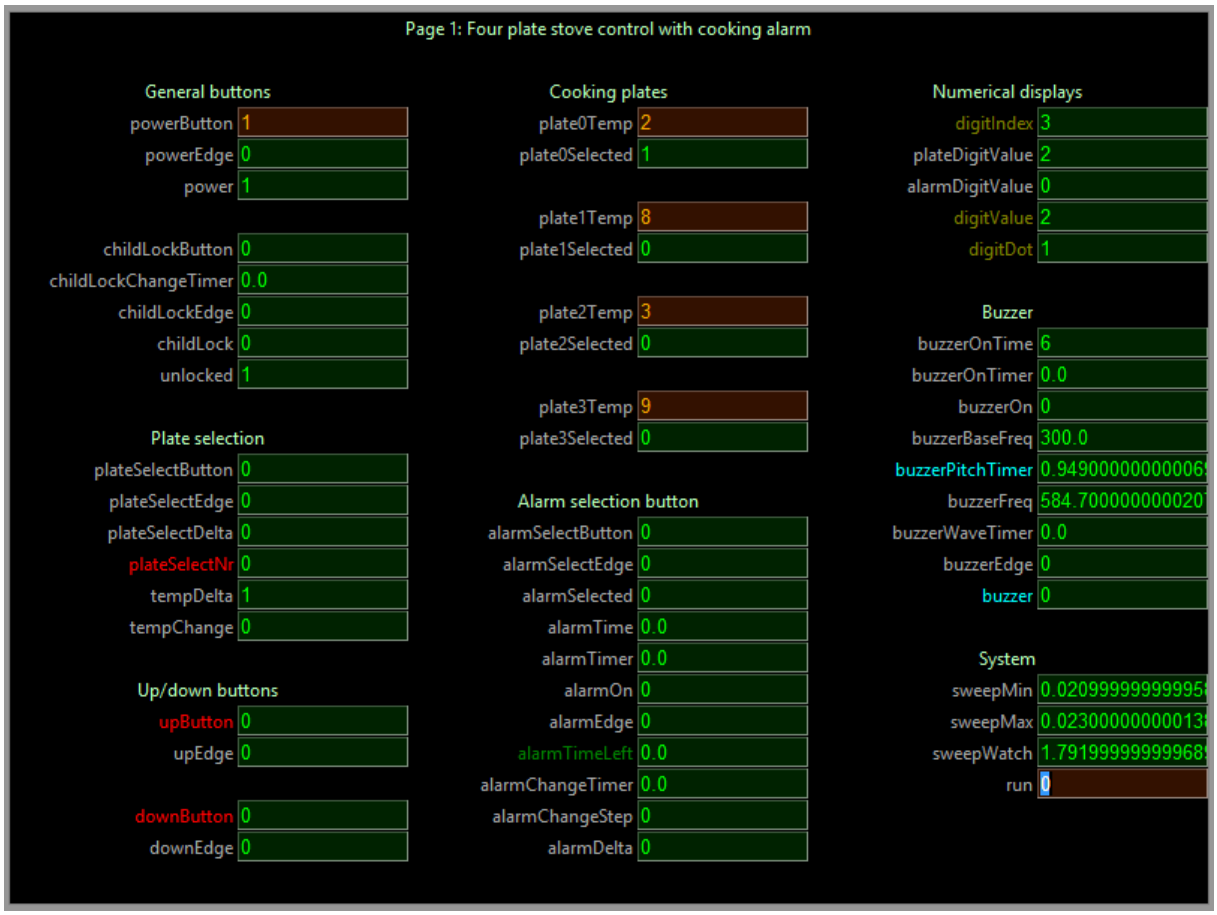

Figure 7. Control panel of the stove example.

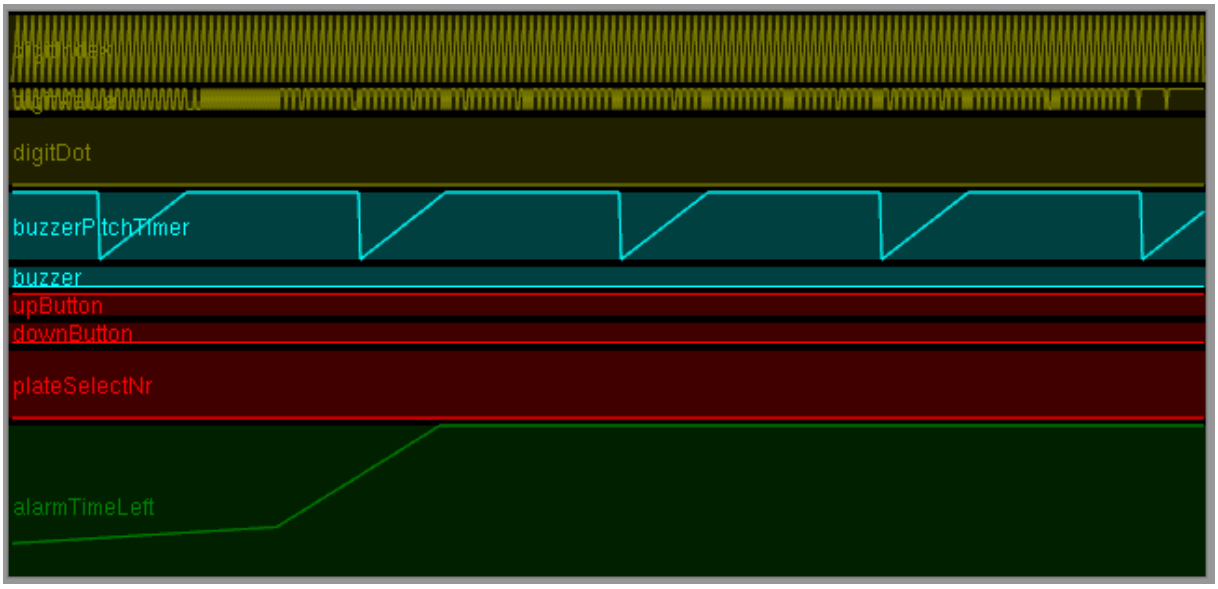

Figure 8. Timing chart of the stove example.

It is interesting to take a good look at *native.cpp* in this case, I/O being not completely trivial due to the multiplexed control of the 7 segment digits.

In general SimPyLC poses no restrictions to what can be done in the *native.cpp* and in any additional C or C++ modules. However, the more simulation is used, the more confident you can be that your software is OK before letting it control any hardware.

While with these toy examples getting the combination of hardware and software to work by trial and error and mere stamina is no big deal, when controlling multimillion dollar hardware in a situation where every delay costs money, simulation has huge benefits. In the stressful, densely populated late hours chaos of putting such systems into active operation, I welcomed every indication that my software was functioning correctly. Of course simulation was only part of the equation. Because...

The simulator/code generator itself can and will contain errors. In accordance with the license it can ONLY be used as an extra check, a great timesaver, but by no means a primary vehicle to achieve safety and security. If everything seems to be OK in the simulation, the next thing to do is desk checking, peer review, general intervision and systematic, professional, safe factory and on-site testing of the final target code that will in fact control the system. Safe in the case of software malfunction that is. Maybe you'll catch 95% of the errors using the simulator, saving months. But you'll have to catch the remaining 5% for sure before relying on your control in any way. After any change, complete regression testing of the final target code is required.

A short story. When a new fighter plane got introduced in the Netherlands, somewhere around the turn of the millennium, a test pilot flew it to the southern hemisphere. As it passed the equator on autopilot it suddenly turned upside down. The pilot took over on manual control and no harm was done. Questions were asked to the avionics software engineer, who looked up from his game console somewhat disturbed since it was his lunch break: O, uh, yeah, tangent is a periodical function, that's true, so something with a minus sign I guess... He's now working in the game industry full time, where he can do no harm.

Some timing hints and pitfalls

Integration and differentiation

The sweeptime of the simulator has a lower bound of ca 20 ms to avoid consuming all processor resources on a multitasking PC that also has other jobs to do. On the Arduino, the sweeptime will be much shorter. It may seem attractive to insert a large *delay* into the *loop* of the Arduino to force a more or less constant sweeptime at a varying processor load, but this is NOT the right way to fix timing. You'll kill the reaction time of your control without necessity.

Also never integrate anything by just adding a fixed quantity per sweep and do not differentiate anything by just computing the difference over two subsequent sweeps. In this way the behaviour of your control is would not be realtime, but depend on the processor load.

Lastly, never use a timer to obtain a fixed delta t over multiple sweeps. This will make your control slow. Use the varying sweeptime (*world.period*) as your delta t instead.

So e.g.

```
accelleration.set   (speed - oldSpeed) / world.period)          # Differentiation

location.set (location + speed * world.period)                  # Integration

oldSpeed.set (speed)
```

You may insert a small *delayMicroseconds* in the main loop in *native.cpp* if the measured sweeptime becomes 0 incidentally, but in most cases this is not necessary. Sweeptime measurement has an accuracy in the order of magnitude of microseconds.

Although more complicated integration and differentiation schemes may be used, the ones above are accurate enough in many practical situations and keep latency to a minimum.

N.B. TIMERS WILL WRAP AROUND IN 50 DAYS OR SO. SO RESET YOUR TIMERS EARLIER THAN THAT. IF YOU NEED TO KEEP TRACK OF LONGER INTERVALS, E.G. RESET THE TIMER EACH DAY AND COUNT THE NUMBER OF DAYS

Edge triggering for buttons and timers

Whenever a pushbutton is used to increment a counter, go to a next step etc., you have to use a oneshot to avoid incrementing or stepping multiple times at only one button press. Failing to recognize this is a common source of problems.

Less clear is the necessity to use a oneshot if you want a timer to trigger certain events. Suppose you want to increment a counter every 3 seconds. The simplest thing would be to use a timer, reset it to 0 after 3 seconds and increment the counter whenever the timer is 0. On the simulator this would work fine. However, SimPyLC timers running on the Arduino have a resolution of 1 ms, which is more than the sweeptime. This means that a timer can remain 0 for multiple sweeps, e.g. incrementing your counter several times. Using a oneshot to edge trigger on your counter becoming 0 will solve this problem.

Making a hierarchical visualistation

Principle

--//--