LPNHE testbench for the LSST sensors
Status report on the Control Software

Claire Juramy, Laurent Le Guillou, Eduardo Sepúlveda

Abstract

We describe the current state of the control software for the LSST sensor testbench at LPNHE. To be able to proceed with all the planned tests and measurements of the LSST sensors, our philosophy was to reuse as much existing code as possible and glue them together using the XML-RPC protocol and some Python scripting. Scripting facilities are critical as many tests have to be automatized, and at the same time testing procedures should be very easy to modify, adapt, deploy and recycle. The remaining question is to find an efficient way to interface the existing software with the LSST CCS: there may be different answers depending of the various ways the LPNHE testbench will be used (sensor studies, LSST CCD production tests, etc).

Contents

1 Brief overview of the bench

The main elements of LPNHE testbench for the LSST sensors are described on fig. 1. This testbench will be used for several CCD tests and sensors studies: the LSST sensors production testing, but also for PTC studies, to quantify charge diffusion, for the optimization of the CCD clocking sequences, and so on. As this bench will be used in several different setups (flat illumination, spots, fringes, etc.), and for many different tasks, its control software needs to be modular and very flexible.

Nearly all the available instruments and devices (except the REB) are connected via RS-232 (or RS-232 over USB) to a Linux box (Ubuntu). The Agilent power supply is controlled through an ethernet link, as well as the New Focus Picomotor motors.

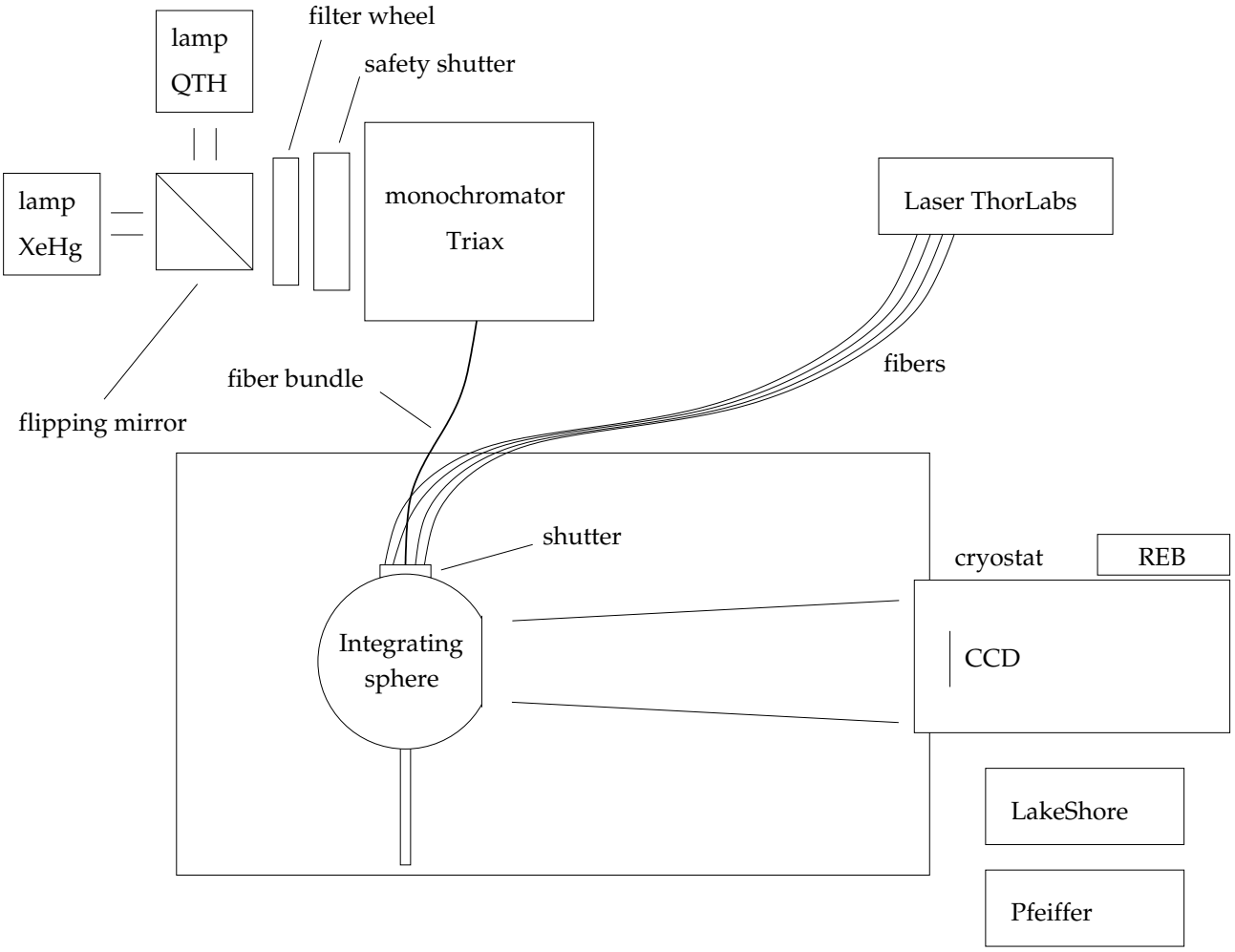

Figure 1: Scheme of the LPNHE testbench for the LSST sensors.

Instrument	link	port	host
Monochromator Triax	RS-232	`/dev/ttyUSB0`	`lpnlsst`
Picoammeter Keithley 6514	RS-232	`/dev/ttyUSB1`	`lpnlsst`
Oriel XeHg Lamp	RS-232	`/dev/ttyUSB2`	`lpnlsst`
Oriel QTH Lamp	RS-232	`/dev/ttyUSB3`	`lpnlsst`
Oriel QTH flux control	RS-232	`/dev/ttyUSB6`	`lpnlsst`
Picoammeter Keithley 6487	RS-232	`/dev/ttyUSB5`	`lpnlsst`
Laser ThorLabs	RS-232	`/dev/ttyUSB8`	`lpnlsst`
Agilent power supply	Ethernet		
New Focus picomotor controller	Ethernet/RS-232		
Filter Wheel (TTL)	NI board		`lpnlsst`
Flipping mirror (TTL)	NI board		`lpnlsst`
Safety shutter (TTL)	NI board		`lpnlsst`

Table 1: Instruments and devices of the bench

2 Control of the testbench instruments

For the large majority of the available instruments and devices on the LSST sensors testbench, we (E. Sepúlveda) already wrote more than one year ago several control panels in C++, with Qt as the underlying library. Qt offers facilities for the GUI as well as the RS-232 serial port. We adapted these existing programs (by adding XML-RPC remote functions) to integrate them with the Python scripting environment.

For a few instruments, we also wrote and recycle native Python drivers.

2.1 Light sources

2.1.1 Lasers ThorLabs

The ThorLabs laser device has four laser diodes with the following wavelengths: 406 nm, 635 nm, 808 nm and 980 nm. Each diode output is sent through a fiber to the integrating sphere. It is controlled by RS-232 (through a RSR-232 to USB converter) and appears on `/dev/ttyUSB8` on `lpnlsst`.

Figure 2: ThorLabs lasers. On the left, the Thorlabs box with its frontend and the fibers connected on the four outputs. On the right, the C++/Qt GUI to control it.

The C++/Qt control panel for this device is launched by:

```
lsstprod@lpnlsst:~> laserthorlabs /dev/ttyUSB8 &
```

The GUI in fig. 2 allows to select the diodes to turn on, and to set up the current for each diode. The "enable" command will activate the light emission once a diode has been selected, and the "disable" to stop light emission (otherwise the 635 nm diode still emits some light).

This program also has an integrated XML-RPC server (port 8082 on `lpnlsst`) in order to control the lasers remotely, for scripting purposes (see section 4).

2.1.2 Lamps

We have two lamps: an halogen (QTH, the default one) and a XeHg lamp. The QTH lamp output can be regulated (but it is important to let the lamp warm before trying to regulate its output). Both lamps are controlled by the `oriel` C++/Qt program (two program instances should be run to control both lamps), launched by the command:

```
lsstprod@lpnlsst:~> oriel /dev/ttyUSB2 &
```

The control panel is shown on fig. 3.

This program has an integrated XML-RPC server (port 8084 for the QTH and 8085 for the XeHg lamp) on `lpnlsst`) in order to control the lamps remotely, for scripting purposes (see section 4).

Figure 3: Left: the two lamps (QTH and XeHg) and the flipping mirror. Right: the lamp control panel for the XeHg lamp.

2.2 Devices controlled by TTL signals: shutters, filter wheel, flipping mirror

Several devices on the sensors testbench are controlled by sending TTL signals: two shutters (at the entrance of the monochromator, and at the entrance of the integrating sphere), the filter wheel before the monochromator, and a flipping mirror to select one of the two available lamps (see 2.1.2). All these devices are connected to a National Instrument board which sends the needed TTL signals. This board is controlled by the following C++/Qt program:

```
lsstprod@lpnlsst:~> ttl &
```

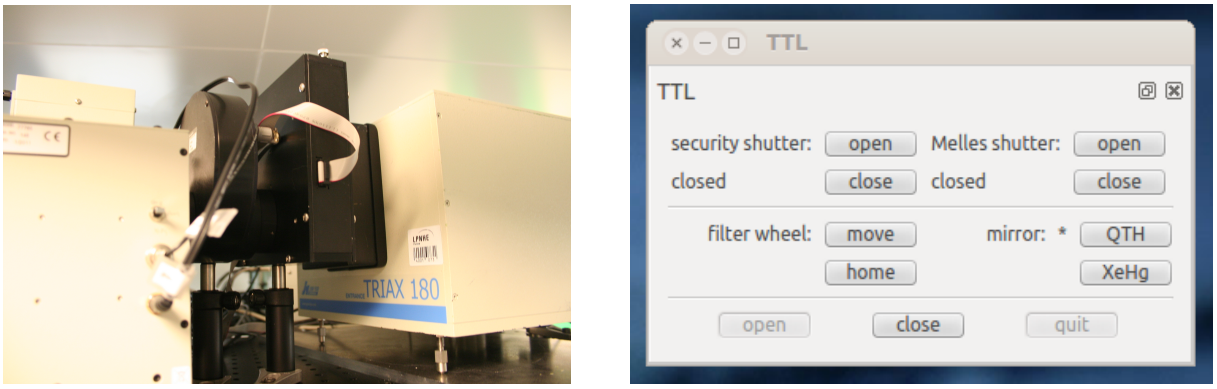

Figure 4: Left: the filter wheel and the safety shutter. Right: the control panel for the TTL signals for the shutters, the filter wheel and the flipping mirror.

For the shutters, the only possible actions are to open or to close them. The filter wheel has a `home` procedure which should be called first. Then each `move` command will rotate the wheel to the next filter position. For the flipping mirror, it is either "on" (QTH) or "off" (XeHg).

Important note: For the CCD tests, in order to have a very precise exposure duration, the shutter mounted on the integrating sphere ("Melles shutter") will be controlled directly by the REB, and not by this program. The other shutter (at the entrance of the monochromator) is only used for safety reasons (the high UV emission from the Hg lamp which is quite dangerous for the eyes) and its timing is not critical.

This program has an integrated XML-RPC server (port 8083 on `lpnlsst`) in order to control the shutters (mainly the safety one), the filter wheel and the mirror remotely, for scripting purposes (see 4).

2.3 Monochromator Triax

The Triax monochromator (fig. 5) has three gratings mounted on a single turret. It is possible to select the grating, select the chosen wavelength, and adjust the width of the entrance and exit slits.

The C++/Qt program to control the monochromator is launched by the command:

```
lsstprod@lpnlsst:~> triax &
```

This software properly manages the dialog with the monochromator controller, by checking that a request action is finished before sending a new request (otherwise the monochromator may block and should be power cycled).

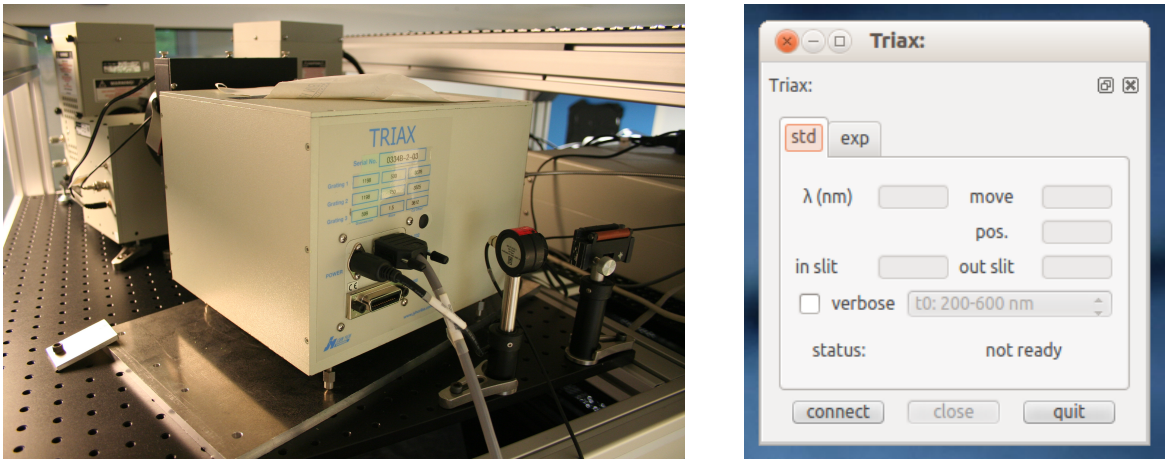

Figure 5: Left: the Triax monochromator. Right: the monochromator control panel.

The `triax` program has an integrated XML-RPC server for scripting purposes (see 4).

2.4 Cryostat instruments

The thermal regulation of the CCD in the cryostat is controlled with a LakeShore. The cryostat pressure is also monitored with a pressure gauge (Pfeiffer); an external electronic device disabled the cooling in case of a vacuum leak. Both instruments are controlled by two C++/Qt programs.

2.5 Motors

2.5.1 XYZ mounting

On this bench we can also project sub-pixel light spots on the CCD sensor, using micrometric holes and a microscope objective fixed on a motorized XYZ mounting, to be able to focus and move the spots (see fig. 6). The XYZ mounting (Pollux linear motors) is controlled by a Python module `lsst.testbench.xyz`, and can also be used directly from the command line with two Python scripts `xyz-position` and `xyz-move`.

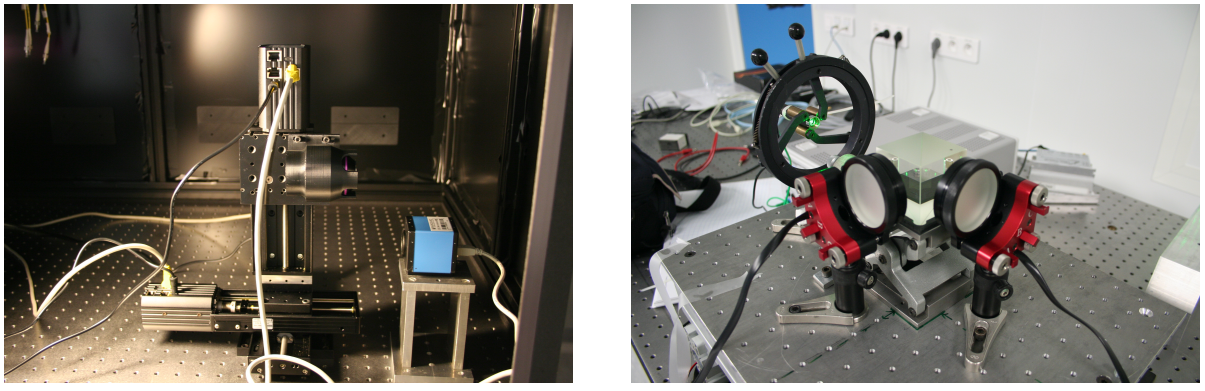

Figure 6: Left: the Pollux XYZ mounting and the Edmund microscope objective to produce subpixel spots on the CCD. Right: the fringe projector.

2.5.2 Picomotor

In order to do MTF measurements on the CCDs (to study the PTC), we use a laser and a Michelson interferometer (see fig 6) to project controlled fringes on the CCD sensor (fringes projection). The precise position and angle of the two interferometer mirrors are controlled by three piezo motors (New Focus Picomotor), for which we wrote two pieces of software:

- A C++/Qt GUI `picomotor` to control the 6 picomotors (mainly to avoid the vibrations generated when we manipulate the mirrors screws by hand);
- A minimal Python module (`lsst.testbench.picomotor`) to automatize the operation of the fringes projector while taking CCD images at the same time.

2.6 Keithley Multimeters/Electrometers

On the LSST sensor testbench we use several Keithley multimeters/electrometers as picoammeter, mainly to measure the current from various photodiodes, in order to monitor the light flux (output of the integrating sphere, incident beam on the CCD). We use different models: Keithley 2000, Keithley 6514, Keithley 6487 (this one also provide the -70 V voltage for the CCD backsubstrate). All of them are controlled by SCPI commands sent through a RS-232 link.

We have two ways to control them.

First, a C++/Qt control panel (shown on fig 7) may be launched for each Keithley multimeter connected:

```
lsstprod@lpnlsst:~> keithley  /dev/ttyUSB1 &
```

This program allows to do sequences of current measurements in various ways and to save them to ASCII data files. For the Keithley 6487 an extra tab in the GUI automatically appears and allows to set up the output voltage and its current limit. It is currently used to provide the -70 V voltage for the CCD backsubstrate.

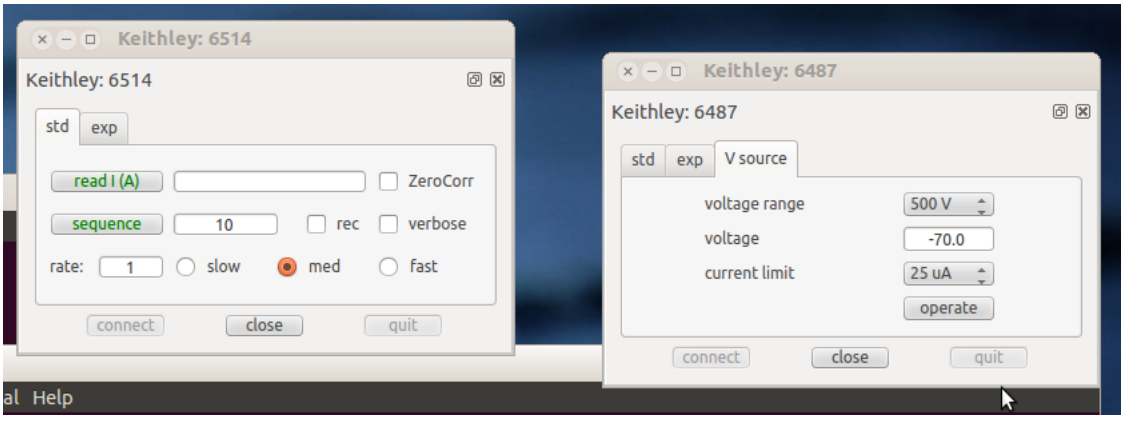

Figure 7: Control panel for two Keithleys picoammeters.

Second, for more specific uses, a Python module `lsst.testbench.keithley` is also used, which offers more flexibility and allows to use any combination of SCPI commands (current, charge, voltage measurements, and so on).

2.7 Small CCD camera for tests

To validate several measurement methods we use a small amateur astronomer CCD camera (Imaging Source DMK 41AU02.AS, visible on fig. 6 on the left picture). Depending of the setup, we control this camera either using a video4linux based C++ code (`v4lcamera`), or through with a very minimal Python module based on the Unicap library (`lsst.testbench.dmk41au02as`).

3 REB slow control

In this section, we describe the various pieces of software we are using for the slow control of the REB and the CCD sensor. These codes are prototypes only aimed for this testbench. It could be interesting to implement some features described here in the CCS (mainly for the sequencer).

3.1 Bash scripts

We (C. Juramy) wrote several Bash scripts to interact with the REB FPGA: they could be used to load the REB configuration (clock sequences, sequencer programing, CABAC voltages) and to do various REB operation like taking a bias, a dark, a CCD image, clearing the CCD, clearing the serial register, and so on.

```
[...]
#CABAC programing
#registers:
#0:OD0EM OD1EM OD0RM OD1RM
#1:IP0 IP1 IP2 IP3
#2:IS0 IS1 IS2 IRG
#3:GD OG RD SH
#4:SL Mux0Mux1 PulseEnaMuxEna00 0

rriClient 2 write 0x500010 0xc5c5c5c5
rriClient 2 write 0x500011 0x00000000
rriClient 2 write 0x500012 0x00000000
rriClient 2 write 0x500013 0xaa008000
rriClient 2 write 0x500014 0x00000000

rriClient 2 write 0x500020 0xc5c5c5c5
rriClient 2 write 0x500021 0x00000000
rriClient 2 write 0x500022 0x00000000
rriClient 2 write 0x500023 0xaa008000
rriClient 2 write 0x500024 0x00000000
[...]
```

These scripts work very well but are not that easy to adapt to various tasks. That is the main reason we develop some more abstract control code.

3.2 Low-level Python interface: PyREB

To simplify the REB slow control and at the same time keep the flexibility needed for our sensor studies, we (L. Le Guillou) wrote a Python module `lsst.camera.reb` for the REB slow control.

3.2.1 REB basic operations

The `lsst.camera.reb` Python module essentially rely on the `rriClient` program to read/write at the REB FPGA subadresses. The low level read/write operations are done by successive calls to `rriClient`:

```
R = reb.REB(reb_id = reb_id)
a = R.read(0x02)
l = R.read(address = 0x20, n=4) # read 4 words
R.write(address = 0x21, 0x3e45)
```

Various methods are provided to do simple operations: launching the sequencer main subroutine (`REB.start()`, `REB.stop()`), getting/setting the internal clock (`REB.time`), getting the FPGA state (`REB.state`), triggering operations, and so on.

3.2.2 Clock sequences

The 16 FPGA clock sequences (“functions”) may be loaded from a XML sequencer file, or created on the fly by the script, like this:

```
func_line_transfer = \
    fpga.Function(name = "line transfer",
                  timelengths = {  0 : 100,  # x10ns
                                   1 : 1000,
                                   2 : 1000,
                                   3 : 1000,
                                   4 : 1000,
                                   5 : 1000,
                                   6 : 1000,
                                   7 : 1000,
                                   8 : 1000,
```

```
                    9 : 0 },
#
#                           ........S...SPPPPRSSSCRRR
#                           ........H...T4321G321LSDU
outputs =      {  0 : 0b00000000000000011010111100,
                  1 : 0b00000000000000111010111100,
                  2 : 0b00000000000000110010111100,
                  3 : 0b00000000000000110110111100,
                  4 : 0b00000000000000100110111100,
                  5 : 0b00000000000000101110111100,
                  6 : 0b00000000000000001110111100,
                  7 : 0b00000000000000011110111100,
                  8 : 0b00000000000000011010111100,
                  9 : 0 } )
```

The 16 “functions” may also be downloaded from the FPGA memory.

These features are needed for the optimization of the clocking sequences: this way, it is possible to take many frames while slightly modifying the clocking scheme before each frame readout, like this:

```
# taking a frame
R.run_subroutine('acq')

# alter the clock sequence for the line transfer (slot 2)
func_line_transfer = R.dump_function(2)
func_line_transfer.timelengths[2] -= 10 # decrease duration of slice #2
# send back the modified function into the FPGA memory
R.send_function(2, func_line_transfer)

# taking a frame
R.run_subroutine('acq')
```

3.2.3 Programing the sequencer

The REB sequencer program may be either loaded from the XML sequencer files (see below), or from a pseudo assembling language like this:

```
[...]
program = """
main:       JSR     acq             repeat(1)
            END

acq:        JSR     clear           repeat(2)
            CALL    func(1)         repeat(10000)
            CALL    func(6)         repeat(2048)
            JSR     read_line       repeat(2020)
            RTS

clear:      JSR     clear_line      repeat(2020)
            RTS

bias:       CALL    func(6)         repeat(550)
            JSR     read_line       repeat(2020)
            RTS

read_line:  CALL    func(2)         repeat(1)     # line transfer
            # read 550 pixels (10 prescan + 512 + 28 overscan)
            CALL    func(3)         repeat(550)
            RTS

clear_line: CALL    func(5)         repeat(1)
            CALL    func(6)         repeat(550)
            RTS
"""
# loading the default sequencer program
R.load_program(program)
```

```
# launching a clear 10 times
R.run_subroutine('clear', repeat = 10)

# taking a bias
R.run_subroutine('bias')

# taking a frame
R.run_subroutine('acq')
[...]
```

Once the program has been loaded, each subroutine may be launched by calling the `REB.run_subroutine(...)` method. An optional `repeat` argument may be provided.

The program in the FPGA memory may also be downloaded (`REB.dump_progam()`) and disassembled.

3.3 Editing Clock sequences: `timeslots`

To ease the modification of the clocking sequences, we (E. Sepúlveda) wrote a graphical editor, named `timeslots` (see fig. 8). The sequences may be loaded, edited and saved in the XML sequencer format.

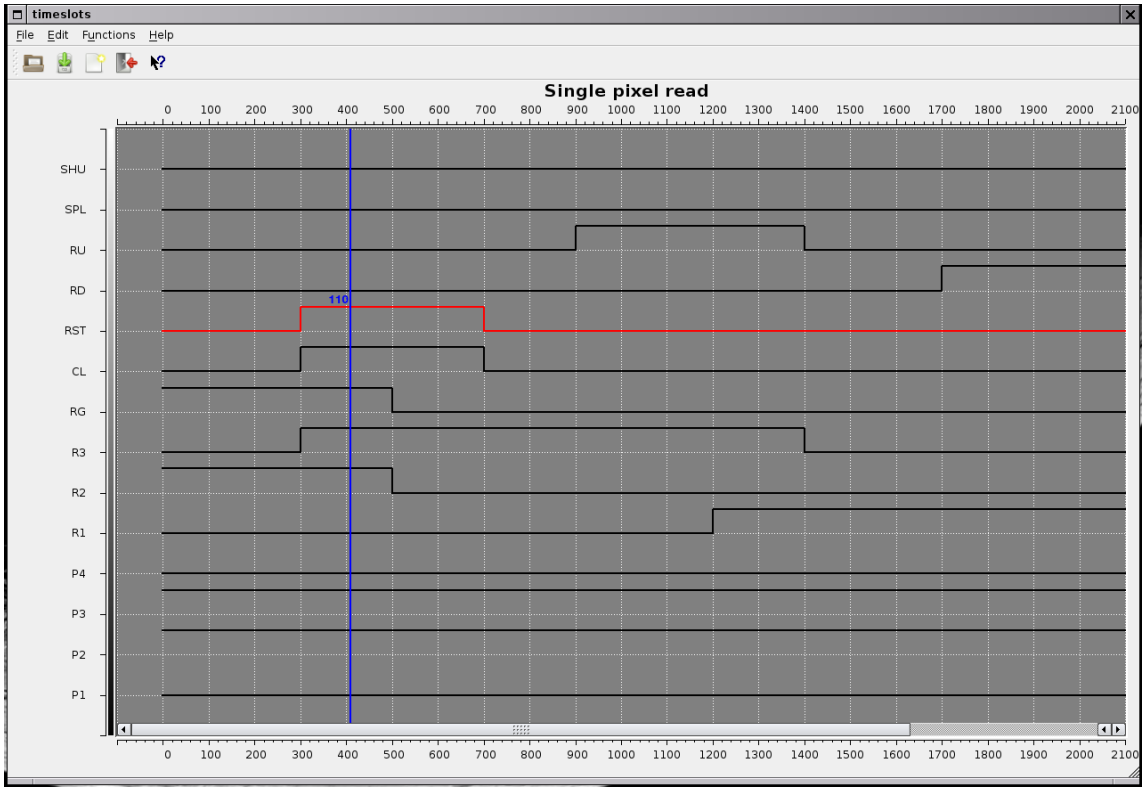

Figure 8: Editor `timeslots` for the clocking sequences.

4 Software integration and scripting

Scripting facilities are critical for the various sensor studies and tests we will run on this testbench. For the moment, our solution is to combine the existing C++/Qt or Python drivers with some Python glue, and use the XML-RPC protocol to remotely control the C++/Qt panels from the scripts. This solution has the advantage that all the instruments do not need to be physically connected to the same computer: this setup is very flexible.

Here is an example of a minimal script commanding the REB and the laser Thorlabs to take flat illuminations:

```
import sys, os, os.path
import time
```

```
import subprocess
import xmlrpclib

import lsst.camera.reb as reb
# import lsst.camera.reb.fpga as fpga

# connect to the laser Thorlabs
laser = xmlrpclib.ServerProxy("http://lpnlsst:8082")
laser.connect()

# Connect to the REB
reb_id = 2
R = reb.REB(reb_id = reb_id)

# Configure the REB
sequencer  = REB.Sequencer.fromxmlfile("sequencer.xml")
# funcs = sequencer.functions
# program = sequencer.program
# R.send_functions(funcs)
# R.load_program(program)
R.send_sequencer(sequencer)

# Compute image size and configure the REB accordingly
# rriClient 2 write 0x400005 0x0010F3D8
R.set_image_size(2020 * 550)

# starting the clock register
R.fpga.start_clock()

# set and print the REB time (could be set to Unix timestamp)
R.time = time.time() / 10.0e-9
print R.time

# starting the imageClient process (requested to receive the frames!)
subprocess.Popen("imageClient %d" % reb_id, shell=True)

# launching a clear 10 times
R.run_subroutine('clear', repeat = 10)

# taking a bias
time.sleep(1)
R.run_subroutine('bias')

# Turn on the laser
laser.select(2) # select one of the 4 laser diodes
laser.setCurrent(100.)
laser.enable()

# taking a frame
R.run_subroutine('acq')

# Turn off the laser
laser.disable()
```