QUICK EXPERIMENT AUTOMATION MADE POSSIBLE USING FPGA IN LNLS

M. P. Donadio*, H. D. Almeida, J. R. Piton
CNPEM / LNLS, Caixa Postal 6192, Campinas - 13083-970, Brazil

Abstract

Beamlines at LNLS are being modernized to use the synchrotron light as efficiently as possible. As the photon flux increases, experiment speed constraints become more visible to the user. Experiment control has been done by ordinary computers, under a conventional operating system, running high-level software written in most common programming languages. This architecture presents some time issues as computer is subject to interruptions from input devices like mouse, keyboard or network. The programs quickly became the bottleneck of the experiment. To improve experiment control and automation speed, we transferred software algorithms to a FPGA device. FPGAs are semiconductor devices based around a matrix of logic blocks reconfigurable by software. The results of using a NI Compact RIO device with FPGA programmed through LabVIEW for adopting this technology and future improvements are briefly shown in this paper.

INTRODUCTION

Experiments at LNLS historically were made using software running in conventional operating systems. Our first software for beamline control centralized all the operations, sensor reading and actuators controlling in a single thread. This was the cause of lots of disturbances in motor movement, when the user moved a mouse or selected a menu item, which almost never ran in a smooth pattern. We could see also a big dead time caused by many tasks running sequentially rather than in parallel.

In 2010 we started to use EPICS [1] and, this way, we could decentralize all the access to the devices. Doing so, we could get many improvements in motor controlling and reduction of dead time. The system became easier to maintain as each device is accessed by a single piece of stan-dalone software. Another advantage was to provide the infrastructure to build a web system named LabWeb [2], based on Science Studio [3] developed and applied at the Canadian Light Source (CLS), that became possible for users to operate the beamline from their universities.

From 2010 until recently in the current year not only the software, but most of the control hardware was replaced in all beamlines. After some years developing, improving and using the new EPICS based system, we faced a new challenge: the beamlines had its photon fluxes and detector sensitiveness increased by the hardware improvement and software was again the bottleneck for some experiments. EPICS can provide a good experiment control using only software and not firmware, in conventional operating systems, if there is no concern about times lesser than 100 ms. Control loops quicker than 10 Hz are not guaranteed to be attended inside a good error limit. To go beyond this limit we needed to change the paradigm adopting a real-time system, for example, or implementing the fast piece of the experiment in hardware.

The first beamline to see this limitation was IMX [4], followed by XRF [5]. Both needed to synchronize motor position, detector shot and shutter in a few milliseconds with error margin of some microseconds.

The third beamline case was SAXS1 [6], whose system is presented in this paper. The development of the system for SAXS1 was used also to test the approach to the beamlines that will be built in Sirius [7], the new Brazilian Synchrotron. The timing for Sirius beamlines needs to be in the microsecond loop or faster.

SAXS1 HARDWARE

We need a system with low jitter between the trigger signal and the start and the end of data acquisition. Going near the hardware is an approach to reduce jitter and that is the motivation to choose FPGA technology. We had available in our inventory a cRIO NI 9144 [8], a unit with FPGA, so this was a natural choice to use in the work. cRIO NI 9144 can be connected to a PXI [9] only by an EtherCAT connection.

Figure 1 shows the connection between the cRIO and detectors and actuators.

There are two main detectors, a Pilatus [10] 300K to measure SAXS and a Pilatus 100K to measure WAXS. Pilatus 300K is configured to collect data in a determined frequency and sends a trigger TTL signal every time an acquisition is being done. cRIO is responsible to send a TTL trigger to Pilatus 100K as it can start acquisition.

Two photo-diodes provide the beam intensity data through a Stanford SR570 [11]. One of the photo-diodes is located in the beam stopper and receives radiation after the sample. The other one is located before the sample and receives the reflection of a small part of the beam thanks to a mylar or kapton thin film positioned in a 45-degree angle related to the beam. cRIO reads the voltage data from Stanford SR570 using a NI 9215 module [12].

cRIO also controls a fast shutter to limit to a minimum the time the sample is irradiated, by opening the shutter only when Pilatus is acquiring.

A cRIO is used to:

- run the FPGA code that sequences the experiment
- read voltage from Stanford using a NI 9215 module

* marcio.donadio@lnls.br

ISBN 978-3-95450-148-9

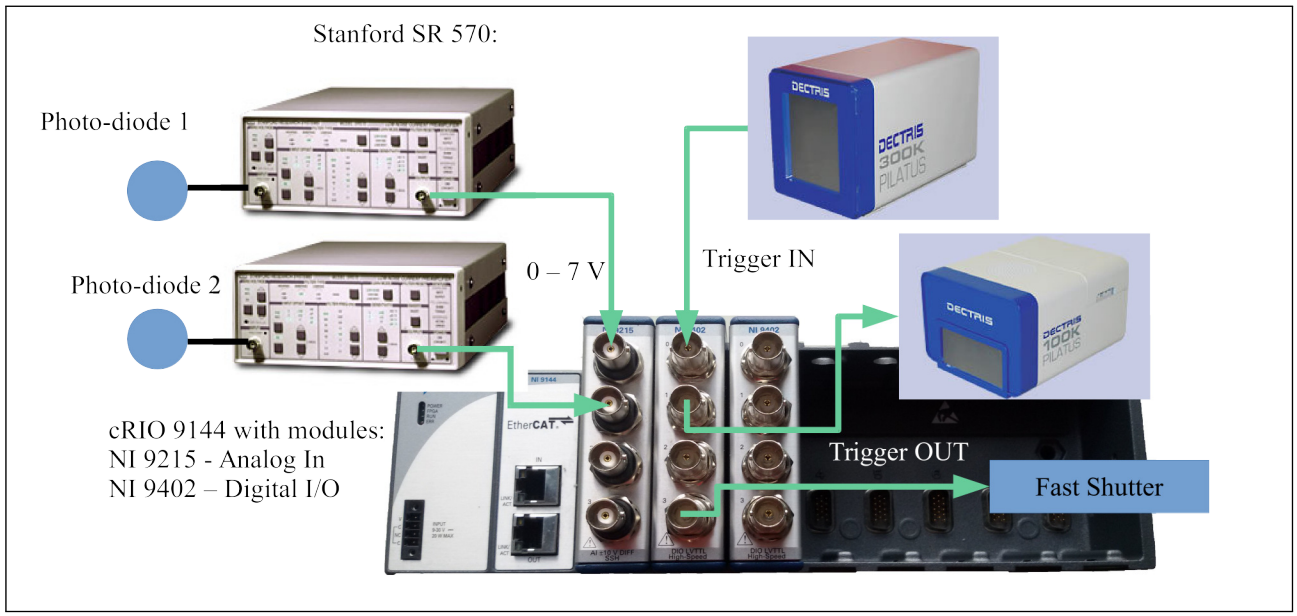

Figure 1: SAXS1 Hardware Connections.

- read trigger signals from Pilatus and to write trigger signals also to Pilatus and to the fast shutter using a NI 9402 module [13].

Figure 2 shows the connection between cRIO, PXI and EPICS clients.

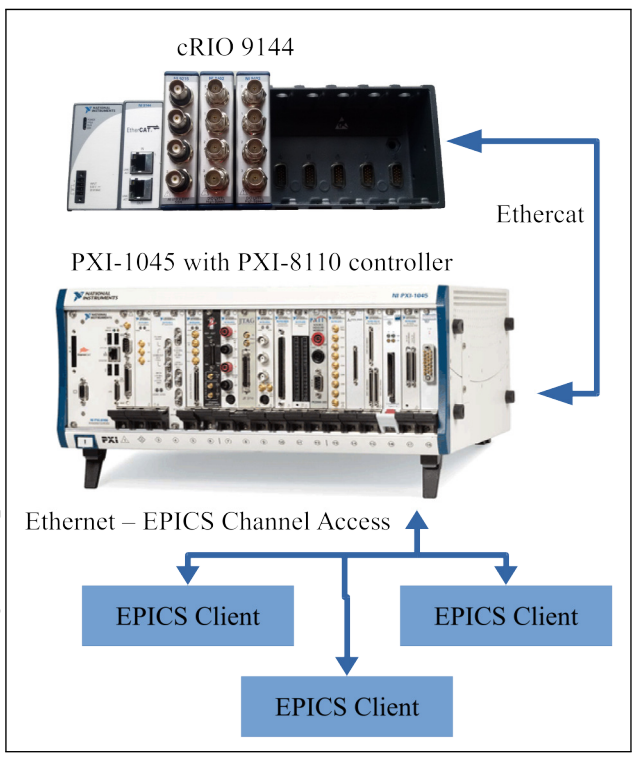

Figure 2: SAXS1 cRIO - PXI connection.

Data collected in cRIO is transferred to the PXI through an EtherCAT connection to be used by a software running in LabVIEW RT operational system inside the PXI controller. Data from LabVIEW RT is passed to the Linux running in the same PXI by Hyppie [14, 15]. In Linux we have an EPICS IOC that sends the collected data to client software by using the beamline network.

FPGA IN cRIO

FPGA in cRIO is programmed using LabVIEW [16]. FPGA code is responsible to control the experiment sequence: receive trigger signal → open shutter → collect voltage from Stanford until trigger signal is low → close shutter → calculate voltage average during the time the trigger signal was high → save the data.

Figure 3 shows the loop where data is collected from the analog input module and summed over the time. When the trigger signal goes TTL low, a division of the sum by the number of loop iterations is made to calculate the voltage average while the detector was acquiring. Not shown in this picture, this average is saved in a FIFO for later use. Let's call this FIFO as FIFO_PDn.

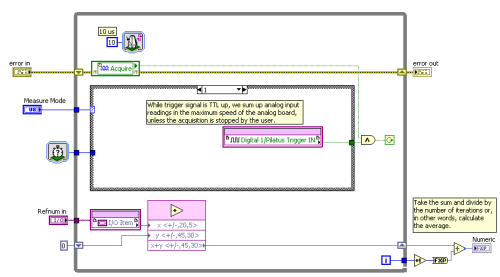

Figure 3: FPGA Voltage Acquisition.

The code is also responsible to administrate data sending over EtherCAT. This is the part of the FPGA code in the deterministic domain. Figure 4 shows a loop where cRIO waits for the rising edge of Scan Engine [17] clock. When

ISBN 978-3-95450-148-9

it arrives, an inner loop gets all data from FIFO_PDn and writes it to a sequence of 50 shared variables per photo-diode, emulating an array. As the Scan Engine speed of cRIO is set to 500 μs in this experiment and 100 data values are sent per EtherCAT packet, we guarantee that the software is able to collect one analog data per photo-diode each 10 μs and send it to the PXI with no data loss. Each data uses fixed point representation with 5 bits for integer part and 20 bits for decimal part. No handshake is implemented because transmission of data over EtherCAT is deterministic and lossless. In fact, we ran millions of tests at the maximum rate and never got a single data loss. Nevertheless it is possible to diagnose when a data is lost and, in this case, the experiment is aborted.

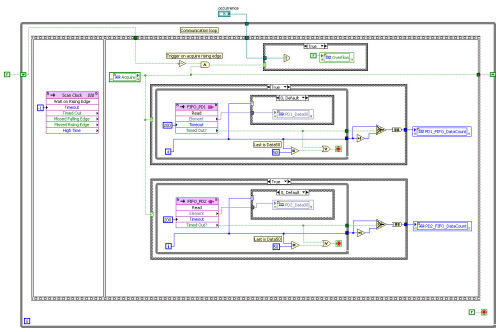

Figure 4: FPGA sends data to LabVIEW RT.

SOFTWARE IN LabVIEW RT

Inside LabVIEW RT there is a LabVIEW software responsible to get information from cRIO and to send it to the shared-memory between LabVIEW RT and Linux. The program collects data using a real-time loop synchronized with the scan engine period (Fig. 5). This is the part of the LabVIEW RT code in the deterministic domain. Doing so it is guaranteed that no data will be lost, as long as the loops fit within the scan engine cycle period, because the EtherCAT cycle is synchronized to the scan engine cycle. In the case of communication failure, experiment is aborted.

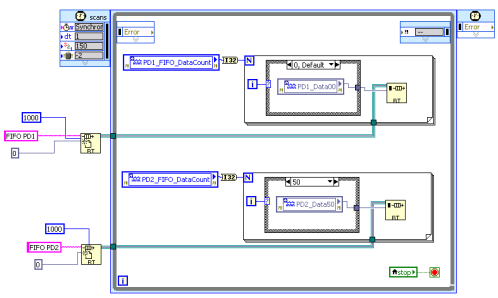

Figure 5: LabVIEW RT software gets data from FPGA.

The collected data is written in an array and is sent to the shared-memory asynchronously. The array is passed to the Linux IOC using Hyppie. The Hyppie shared-memory structure used is described in Table 1 and 2. The tables show the description of each memory address and points which software is responsible to write in it: software running in LabVIEW RT or the IOC running in Linux.

Table 1: Shared-memory Structure - Part 1

Addr	Description	Written by
0	Command	RT / Linux
1	Status message (0 = stopped, 1 = running)	RT
2	Last error found	RT
3	Current measurement point	RT
4 - 13	Current amount of data read from cRIO Diode 1 - 10	RT
14	Delay time before each frame acquisition (in ms)	Linux
15	Delay time after each frame acquisition (in ms)	Linux
16	Number of points to measure (maximum 1000)	Linux
17	Trigger from? (1 = Pilatus, 2 = environment)	Linux
18	Close Shutter	Linux

Table 2: Shared-memory Structure - Part 2

Addr	Description	Written by
19 - 28	Type photo-diode 1 - 10 (1=Stanford,2=Keithley)	Linux
29	Dark current time(in ms)	Linux
30 - 39	Dark current value from photo-diode 1 - 10	RT
40-1039	Measurement array photo-diode 1	RT
1040-2039	Measurement array photo-diode 2	RT

SOFTWARE IN LINUX

In Linux we developed some device supports [18] that read data from the shared-memory, using Hyppie to access memory addresses described in Tables 1 and 2. This IOC provides PVs [19] to configure, start, stop and read parameters from the experiment sequencer programmed in cRIO's FPGA. To access these PVs we are using Py4Syn [20] and CS-Studio [21] as EPICS clients.

RESULTS AND DISCUSSION

Figure 6 shows on the top the trigger signal that is sent to the system, with 1.25 μs high and 20 μs low, resulting in

ISBN 978-3-95450-148-9

a total period of 21.25 μs. On the bottom we can see the signal that indicates when FPGA stopped to read the voltage signal and detects that the trigger signal was low. We could run all the algorithm in FPGA cyclically with a worst case period of 21.25 μs (see the red square in Fig. 6). In the best case we could get 13.6 μs, as seeing in the first trigger pulse shown in Fig. 6. The acquisition time of NI 9215 module is approximately 10 μs and we read it inside a loop. The time difference between the best and the worst case happens when the trigger signal goes low in some point of the start of this loop. So FPGA runs another complete acquisition time before detecting that the measurement needs to be stopped.

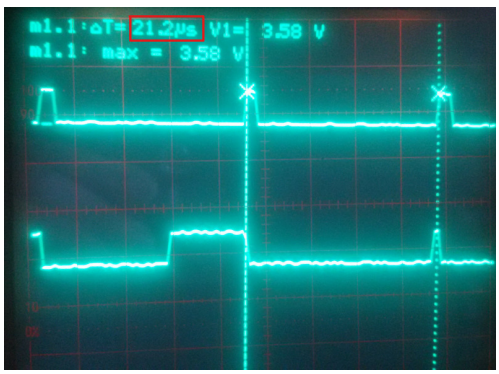

Figure 6: Time to detect trigger unset signal.

CONCLUSION

This entire system is used in simple SAXS and WAXS acquisitions, using or not chemical kinetics or any other complex experiment using synchronization of the system with the sample, as temperature and tensioning forces acquisitions with good stability. The period of 21.25 μs outranges the frequency needed by SAXS1 beamline - 20 Hz - as the fastest image acquisition that can be done is 50 ms, due to available x-ray flux. Even with sufficient flux, the next bottleneck would be the fast shutter that opens in 25 ms and closes in 15 ms.

Despite that, the presented system was built to be prepared for new challenges in Sirius. In fact, the new Coherent and Time-Resolved Scattering beamline that will be built in Sirius, named CATERETE, will study fast kinetics in microfluidic and the beamline will need a few microseconds in resolution. So we could say that the architecture presented here is a strong candidate for Sirius beamlines control solutions. Nevertheless, we will need to build faster systems to achieve our new nanoseconds frontier and keep the work that was started with the presented system.

ACKNOWLEDGMENT

The authors want to acknowledge SAXS1 beamline staff Florian Edouard Pierre Meneau, Tiago Araújo Kalile and Carolina Vieira Comin who tested this system over and over, adding valuable comments and suggestions to improve reliability, correctness and precision of the solution.

REFERENCES

[1] EPICS website: `http://www.aps.anl.gov/epics/`

[2] H. H. Slepicka *et al.*, "LabWeb – LNLS Beamlines Remote Operation System", TUPPC037, Proc. ICALEPCS 2013

[3] N. Sherry *et al.*, "Remote Internet Access to Advanced Analytical Facilities: A New Approach with Web-Based Services.", *Analyt. Chem.*, 2012, Vol. 84 (17), p. 7283–7291

[4] G. B. Z. L. Moreno *et al.*, "On-the-Fly Scans for Fast Tomography at LNLS Imaging Beamline", *these proceedings*, THHB3O03, ICALEPCS 2015, Melbourne, Australia (2015).

[5] XRF description: `http://lnls.cnpem.br/beamlines/xafs/beamlines/xrf/`

[6] SAXS1 description: `http://lnls.cnpem.br/beamlines/saxs/x-ray-beam/`

[7] L. Liu *et al.*, "Update on Sirius, the New Brazilian Synchrotron Light Source", MOPRO048, Proc. IPAC2014

[8] NI 9144 Expansion Chassis Under the Hood (2012)

[9] What is PXI? (from NI)

[10] Hybrid Photon Counting Detectors for Your Laboratory (from Dectris)

[11] MODEL SR570 Low Noise Current Preamplifier Manual (from Standford Research Systems)

[12] Operating Instructions and Specifications NI 9215 (from NI)

[13] Operating Instructions and Specifications NI 9402 (from NI)

[14] J. R. Piton *et al.*, "Hyppie: a Hypervisored PXI for Physics Instrumentation under EPICS", MOPG031, Proc. BIW12

[15] J. R. Piton *et al.*, "A Status Update on Hyppie - a Hypervisored PXI for Physics Instrumentation under EPICS", TUPPC036, Proc. ICALEPCS 2013

[16] LabVIEW System Design Software (from NI)

[17] Using the NI Scan Engine (ETS, VxWorks, Windows) (from NI)

[18] Basic EPICS Device Support, M. Davidsaver: `https://pubweb.bnl.gov/~mdavidsaver/epics-doc/epics-devsup.html`

[19] EPICS Channel Access Overview, K. Kasemir, p. 5 - 6: `https://ics-web.sns.ornl.gov/kasemir/train_2006/1_3_CA_Overview.pdf`

[20] H. H. Slepicka, *et al.*, "Py4Syn: Python for Synchrotrons", *J. Synchrotron Rad.* 22, pp. 1182-1189 (2015).

[21] Control System Studio: `http://controlsystemstudio.org/`

ISBN 978-3-95450-148-9