Common
Electronic Logbook
XES Common
LUSI Common
XTOD
Supported Control & DAQ Devices
Instruments
AMO
CXI
XPP
XCS
LUSI Diagnostics
SXR
MEC
CAMP
EBIT
Detectors
CXI 2D Detector
XPP-2D-Detector
pnCCD-Detector
Subsystems
Networking
Data Acquisition
Data Management
Controls
Laser Control
Accelerator Timing System
Femto-Second Timing
Test-Stands
Personel Protection System
Machine Protection System
XES Laser Safety Systems
User Safeguards
Tools
Confluence How-To
Jira How-To
Internal
Support
Controls How-To
[Controls Problems & Solutions]
Misc
Misc
Page Operations
Browse Space

Photon Control and Data Systems

Online Analysis and Monitoring

Added by Christopher O'Grady, last edited by Christopher O'Grady on Aug 26, 2009 ()
Labels:

Overview

In this page, we will be discussing ways to view the data gathered by the PDS (Photon Data System). This breaks down into two categories, analysis and monitoring.

Analysis refers to code that is linked with PDS code ("pdsdata" framework) and reads archived files. A program called xtcreader is an example that reads data from a file and parses it.

Monitoring refers to code that may be built in any other build system by including a few files from the pdsdata framework. Monitoring code communicates with the PDS system via shared memory. We provide a base class that will be included in the monitoring build to handle the shared memory interface. We also discuss below a development aid that will simulate this shared memory by reading data from a file and feeding it to the shared memory.

Note that the code on the client side can be built in 32 bit and 64 bit versions. We will illustrate both.

File Based Analysis

To do online file-based analysis perform the following steps on a linux box:

1. Download this tar file: lcls data file
2. tar -xvfz release.tar.gz
3. cd release

To build and test the 32 bit version do the following

1. gmake i386-linux
2. build/pdsdata/bin/i386-linux/xtcreader -f opal1k.xtc
3. build/pdsdata/bin/i386-linux/xtcreader -f pnCCD_acqiris.xtc

To build and test the 64 bit version do the following

1. gmake x86_64-linux
2. build/pdsdata/bin/x86_64-linux/xtcreader -f opal1k.xtc
3. build/pdsdata/bin/x86_64-linux/xtcreader -f pnCCD_acqiris.xtc

There are two important requirements:

- you need gmake 3.81 or later
- you need to run on a little-endian machine (read: intel processor)

You can see the output from doing the above here.

- the opal1k.xtc file was generated by the real LCLS DAQ system. it contains events with Opal1000 data.
- the pnCCD_acqiris.xtc file is a real LCLS DAQ system file containing Acqiris data accompanied by pnCCD camera frames

xtcreader is a simple analysis program that iterates through these xtc files. There are two important ideas that will be needed by most users:

- At the beginning of the file there is the "configuration" information for each device that is pacakged in a C++ class. For the acqiris configuration data this class is accessible to the user in the following callback in xtcreader.cc:

```
void process(const DetInfo&, const Acqiris::ConfigV1&);
```

- For each LCLS shot the detector data is similarly packaged in a C++ class. this should be used together with the configuration data to produce physics results. The following callback gives access to the acqiris waveforms:

```
void process(const DetInfo&, const Acqiris::DataDescV1&);
```

These callbacks are the places where the user would put in calls to their analysis code.

The data structures for file-based analysis are identical to those for online monitoring, as described below, so the same analysis code can be used.

Online Monitoring, and Simulation Using Files

The online monitoring system will interface the DAQ system to the monitoring software through a shared memory interface. The two sides communicate with each other via POSIX message queues. The DAQ system will fill the shared memory buffers with events which are also known as transitions. The DAQ system notifies the monitoring software that each buffer is ready by placing a message in the monitor output queue containing the index of the newly available event buffer.

When the monitoring software is finished with a shared memory buffer it releases the buffer by returning the message via its output queue. The DAQ monitoring system does not store events. If no shared memory buffers are available, then any events that go past the monitoring station during that time will not be monitored.

To facilitate the development and testing of monitoring software, SLAC has developed a file server that mimics the DAQ system. Instead of live data from the DAQ system, it reads a file and presents it to the monitoring software via the shared memory interface the same way that the DAQ system does. One difference is that the file server will not drop events if the monitoring software takes too long. It will wait indefinitely for a free buffer for the next event in the file.

To use the example files you will need two shells. The executables are in release/build/pdsdata/bin/i386-linux-dbg. Assuming both shells are in the executable directory, first start the server with something like:

```
./xtcmonserver -f ../../../../opal1k.xtc -n 4 -s 0x700000 -r 120 -p yourname [-l]
```

This points the software at the xtc example file, specifying four message buffers and the size of each message buffer. The -r option specifies the rate that events will be sent in cps, limited only by the I/O and CPU speeds of the platform you are running on. The last parameter shown is a "partition tag" string that is added the name of the message queue, to allow multiple people to use this mechanism without a "name collision". If only one person is using a machine, then just supply a placeholder name.

If there is more than one user on the computer you are using, you can use the partition tag parameter to resolve conflicts. If you use your login name as the partition tag, then you will be guaranteed not to collide with anyone else.

The optional argument, "-l", will cause the server to loop infinitely, repetitively supplying all the events in the file.

The buffers referred to above are in the shared memory. If you don't need the shared memory to add any latency tolerance then you can use a smaller number. Using only one buffer will serialize the operation because only one side will be able to read or write at a time. The minimum number of buffers used should be two. The buffer size must be large enough for the largest event that will be handled.

The server will load the shared memory buffers with the first events in the file and then wait for the client to start reading the events.

Once the server is started, you can start the client side in the second shell with:

```
./xtcmonclientexample -p yourname
```

The only parameter that must be given to the client software is the partition tag. This must exactly match the partition tag given to the server side software.

Once running the server will keep supplying the client with events until it runs through the file. It will then exit, unless the optional "-l" command line parameter was given. If the looping option is given on the command line, then the server will endlessly repeat the sequence of L1Accept (Laser shot) events in the file and will not terminate until the process is killed.

The client has no way of knowing when it's at the end, so you must kill the client to terminate it. If you restart the server without having killed the client, they won't talk to each other.

Sample output from running the above can be found here.

To write your own monitoring software, all you have to do is subclass the XtcMonitorClient class and override the XtcMonitorClient::processDgram() method to supply your own client side processing of the DataGram events in the XTC stream or file supplied by the file server. Below is the example implemented by XtcMonClientExample.cc.

```
class MyXtcMonitorClient : public XtcMonitorClient {
  public:
    virtual void processDgram(Dgram* dg) {
      printf("%s transition: time 0x%x/0x%x, payloadSize 0x%x\n",TransitionId::name(dg->seq.service(
             dg->seq.stamp().fiducials(),dg->seq.stamp().ticks(),dg->xtc.sizeofPayload());
      myLevelIter iter(&(dg->xtc),0);
      iter.iterate();
    };
};
```

This method is the callback from the client monitoring when it receives a buffer in the shared memory. The example above uses an XtcIterator subclass to drill down into the data and label it. You can provide your own functionality there instead.

Children (2)

Analysis Sample Output
Monitoring Sample Output