myana user guide

In this page:

Introduction

LCLS Data Analysis frameworks are under development. The currently supported approaches for analysis of LCLS data are:

- **myana** — A C++ program to analyze an xtc file. Provided (and used) by the DAQ group. How to set up your own myana executable is explained in the DAQ section "A Simple Online Analysis Example" .
- **pyana** — A python-based analysis framework. Pyana User Manual
- **PSAna** — A C++-based analysis framework. Psana User Manual - Old

This document attempts to explain the names and functions found in the myana code and give some working examples on how to set up your analysis software. And we try to explain the structure of the data file and how to extract useful information from your data.

In several of these examples, we fill root histograms or NTuples. For more information on root, see http://root.cern.ch.

If you have questions or requests related to this user guide, feel free to send me an email (ofte at slac.stanford.edu).

The data file format: xtc

The data recorded from the LCLS experiments are stored in xtc (eXtended Tagged Container) files. This online format consists of "datagrams", structures that have fileds like *TypeId*, *Damage*, source (*Src*) and *extent* (size). Xtc files are not indexed and does not provide random access. The data file contains only data, no metadata, so you depend on the pdsdata library (or similar) to make sense of the files. The only way to read it is using a special iterator (XtcIterator) and read the events seqencially, one shot at a time. The *myana* application does this loop for you, and you can customize the `event()` function to read out the information you want from the xtc file.

You can explore the contents of an xtc file by using the `xtcreader` or `pyxtcreader` utilities:

```
pslogin ~ >  xtcreader -f myxtcfile.xtc | less
pslogin ~ >  pyxtcreader  myxtcfile.xtc | less
```

Reading through the output, you may see sections describing the various transitions in datataking. Look for these "headings" in the text output:

- Configure transition
- BeginRun transition
- BeginCalibCycle transition
- Enable transition
- L1Accept transition — This is the event data. Each event starts with "L1Accept transition:". From the text that follows, you can get an idea of what detector data is in the xtc file.

A new tool to list the contents of the file:

```
pslogin ~ >  xtcsummary.py
```

lists all detector and epics information found in Configure and L1Accept (event data) sections, and lists number of events per calibration cycle. Can be helpful when putting together your myana or pyana analysis script. Example output.

You can analyze the xtc data with the offline tools, `myana` and `pyana`. You also have the option of using the *hdf5* data format (hierarchical data format 5), but you will have to wait for the *xtc -> hdf5* translation which may take some time. Also, there is no support for *hdf5* analysis by the offline group quite yet. *hdf5* will be the standard offline LCLS data format, and tools are under development for analyzing these files. More about data formats and where to find the experiment data files, see Analysis Workbook. Data Formats

The `pdsdata` library

See also pdsdata Reference Manual.

Myana uses the pdsdata library to access the datagrams in the xtc files, thus in this context `pdsdata` defines the data structure. *pds = photon data system*. In your analysis directory you'll find it in `release/pdsdata/`. The header files are in the top level directories of each package, and the implementation files are in the src directory of each package. Here's very briefly what the library contains:

package	description
`ipimb`	Intensity position, intensity monitor board (IPIMB) Four diodes positioned around the beam measure scattered X-rays. Based on the output voltage from the four sensors, we can determine pulse intensity and position of the beam. Note, that the lusi package contains methods to get feature-extracted / background subtracted output from the IPIMB.
`encoder`	SXR SLE Info (Laser Mirror Position Encoder)
`pnCCD`	for the two CCD detectors used by the CAMP collaboration
`acqiris`	DAQ interface to the Acqiris digitizer hardware. Waveform data.
`camera`	General structure to read camera frames, configurations, feature extracted info
`evr`	EVent Receiver (event code / beam code)
`opal1k`	Specialized interface for Opal1000 camera. Depends on the camera `package`
`pulnix`	for Pulnix TM6740CL monochrome camera used to read out the YAG screens
`control`	utility for DAQ control, PV (process variable) control and monitoring
`xtc`	This package defines all the datagrams for the xtc file.
`epics`	DAQ interface to epics (process variables (PV))
`bld`	DAQ interface to BeamLine Data, e.g. FeeGasDetEnergy, EBeam, PhaseCavity
`princeton`	DAQ interface to the Princeton camera
`fccd`	LBNL/ANL Fast CCD monochrome camera
`cspad`	CXI CsPad detector
`lusi`	LCLS Ultrafast Science Instruments Configs for diode, ipm, pim.
`app`	Xtc and Epics readers

myana.cc .... an example C++ program to extract information from xtc file

This example fetches data for each event and writes it to a root histogram and stores the histogram in a root file. You may want to store your data differently, e.g. one histogram for each event, or everything in a root ntuple for further processing. Or you can write some other format that you'd like to work with (ascii file, ...).

myana.cc - example code that makes a simple averaging histogram
main.cc - defines the functions used by myana.cc

myana_morefeatures.cc - example code that does a little more than myana.cc
examples/myana_cspad.cc - example code to read out data from the CsPad XPP detector.

The examples above are meant to show you how you can make your own code. With different experiments using different hardware and having different goals, these examples might not apply to your particular experiment / datafile. If so, you'll need to search the main code and libraries a bit to find something more suitable. Here's a brief description of the functions of the myana.cc example and main.cc:

myana.hh and **myana.cc**:

This is the "user analysis module". This is where you fill in your own code to extract the information that *you* want from your experiment's xtc file. This module contain only the following functions:

```
beginjob()   // called at the beginning of an analysis job. You can for instance book histograms here.
beginrun()   // called at the beginning of a run (the analysis job might analyze several runs)
begincalib() // called for each calibration cycle
event()      // this is where you fetch, process and store information about each event (shot).
endcalib()
endrun()
endjob()
```

In the example, a profile histogram is booked in `beginjob()` and voltage vs. time is filled in each event. The profile histogram displays the average value of all events.

main.hh and **main.cc**

This is the main control of the analysis, but you should avoid editing this file. When all the utility functions (in main) and user functions (in myana) have been read, `main()` is executed and controls the flow of the analysis. For each xtc file it calls

```
anafile(xtcname, maxevt, skip, iDebugLevel);
```

which uses a special iterator to loop through all the datagrams in the file, and makes sure to execute the `beginjob()` and `event()` functions that you implemented in myana.cc.

All the functionality needed to get data from the xtc file is (or should be) defined in main.cc and in the files it includes (including the pdsdata library). You just need to call the appropriate functions from your myana.cc to extract the information you need from the file. Get an uppdated list of all the available functions by looking at main.hh (implementations are in main.cc).

More examples

- **myana_morefeatures.cc**
  This version of the "user analysis module" shows how to obtain some more information from the xtc file:
  - `beginjob()`:
    - we book a profile histogram for AMO Ion Time-of-flight (AmoITof) waveform data, and also five regular histograms to fill with single event data from the first five events. To do this we need some information about the AmoITof configuration, which is obtained using the getAcqConfig(). This gives us the number of channels that were used, number of samples and sampling intervals, all needed to book the histogram.
    - also a constant-fraction histogram is booked for AmoITof. This has it's own fill function, as we shall see from the event() function.
    - For the Electron Time-of-flight detector (AmoETof), we similarly get the configuration data and make one profile histogram for each channel used.
    - Also get config information about the Magnetic bottle electron spectrometer (AmoMbes).
    - A Princeton camera and a fast CCD (FCCD) was also in use. These have their own getConfig functions: getPrincetonConfig(DetInfo::SxrBeamline, ...) and getFccdConfig(SxrFccd, ...).
      In beginrun() we get the config info from AmoITof again, to check if it changed between runs in the same job.
  - `event()`:
    - fills the histograms booked at the beginning of the job: getAcqValue() gets the data from a given detector for each event. The main program is already keeping track of which event we're processing at the time. The constant-fraction histogram is filled by the function fillConstFrac(), defined in main.cc. This histogram is filled with the boundary position each time the pulse crosses the threshold,
    - the rest of `event()` uses a lot of get-functions to show how to use some of these. Generally, they all give you values through scalar or array variables passed as arguments to the functions. The example doesn't show what you would use this information for, but you might already know that 🙂

- myana_etof.cc
  More histogram building for ETof Acquiris

- myana_itof.cc, myana_bin.cc
  More ITof Aquiris averaging, and more about binning

- myana_mbes.cc
  Magnetic electron bottle spectrometer (Mbes) Acquiris, time resolved binning

- myana_esort.cc, myana_bin.cc
  Energy binning for Mbes Acquiris

- ACQexp.cc, EXSavg.cc, (or EXSavgOMP.cc for parallel processing)
  Opal image processing, projections, image export

- **examples/myana_tuple.cc**
  Example of how to store several variables in a root NTuple for further processing (histogramming, correlation studies etc.).

- **examples/myana_cspad.cc**, **examples/CspadTemp.cc**, **examples/CspadTemp.hh**
  CsPad image

Configuration and Event Data retrieval functions:

The following contains a few lines of explanation for some of the functions defined in main.
But first some general remarks:

- Most of the functions return 0 if it was a successful function call, any other number means it failed.
- Values are obtained through the arguments of the function calls. E.g. declare an array in your myana.cc, and `getXXXValue(&myarray[0])` will fill the array for you.
- Enums: Several of the functions can be used to extract data from several of the detectors. Which detector is specified by an enum (named constant integers). You are encouraged to use the names instead of the numbers, in case the underlying order changes in a new version of the program.

Acquiris digitizer

```
int getAcqConfig(AcqDetector det, int& numChannels, int& numSamples, double& sampleInterval);
```

Fetches the configuration information for any of the Acquiris devices. Returns 1 if the requested detector does not exist, and 2 if it was not in use. Tells you the number of channels used for this device, the number of samples collected and the sample interval. This is typically done in the `beginjob()` or `beginrun()` functions.

```
int getAcqValue(AcqDetector det, int channel, double*& time, double*& voltage);
int getAcqValue(AcqDetector det, int channel, double*& time, double*& voltage, double& trigtime);
```

Fetches waveform data from any of the Acquiris devices. Fills your arrays with the waveform time and voltage, and optionally gives you the trigger time. This should be called from within the `event()` function.

In the myana.cc example, we fetch data from the AmoITof device (AMO Ion Time-of-flight).
Other Acquiris devices (see main.hh for an up-to-date list):

```
AMO:
   AmoIms      - ion momentum spectrometer (2 detectors, 7 channels)
   AmoGasdet   - gas detector (in the Front End Enclusure)
   AmoETof     - electron time-of-flight (5 detectors)
   AmoMbes     - magnetic bottle electron spectrometer
   AmoVmiAcq   - (Vmi = Velocity map imaging)
   AmoBpsAcq   - (Bps = Beam position screen)
   Camp        - for the CAMP experimental setup
SXR:
   SxrBeamlineAcq1
   SxrBeamlineAcq2
   SxrEndstationAcq1
   SxrEndstationAcq2
```

Image data

There are several getters for fetching image data from the xtc file. Depending on which camera was in use, one of these should be appropriate:

- Opal1000 camera:
  `getFrameValue` (an alias getOpal1kValue is provided for backward compatibility)
- Pulnix6740CL camera:
  `getFrameValue` (an alias getTm6740Value is provided for backward compatibility)
- FrameDetector (general):

  ```
  int getFrameConfig   (FrameDetector det);
  int getFrameValue(FrameDetector det, int& frameWidth, int& frameHeight, unsigned short*& image );
  ```

  Gives you the width and height (in pixels) of the image, and a pointer to the start of the pixel array of a Pds::Camera::FrameV1 object. Specify the detector (using an appropriate enum).

```
Available frame detectors:
  AMO:
      AmoVmi        - velocity map imaging
      AmoBps1       - beam position screen
      AmoBps2       - beam position screen
  SXR:
      SxrBeamlineOpal1
      SxrBeamlineOpal2
      SxrEndstationOpal1
      SxrEndstationOpal2
      SxrFccd
   XPP:
      XppSb1PimCvd
      XppMonPimCvd
      XppSb3PimCvd
      XppSb4PimCvd
```

- XPP CsPad detector:

```
namespace Pds { namespace CsPad { class ConfigV1; }}
int getCspadConfig (Pds::DetInfo::Detector det, unsigned& quadMask, unsigned& asicMask);
int getCspadConfig (Pds::DetInfo::Detector det, Pds::CsPad::ConfigV1& cfg);

namespace Pds { namespace CsPad { class ElementV1; }}
int getCspadQuad  (Pds::DetInfo::Detector det, unsigned quad, const uint16_t*& pixels);
int getCspadQuad  (Pds::DetInfo::Detector det, unsigned quad, const Pds::CsPad::ElementV1*& data);
```

Gives you a pointer to the first position in the array of pixel data from the XPP CsPad detector (or alternatively you can get a pointer to the Pds::CsPad::ElementV1 object itself).
For an example of how to draw an image as a 2D root histogram, see the myana_cspad.cc example.

- Fast CCD camera:

```
int getFccdConfig(FrameDetector det, uint16_t& outputMode, bool& ccdEnable,
                  bool& focusMode, uint32_t& exposureTime,
                  float& dacVoltage1, float& dacVoltage2, float& dacVoltage3, float& dacVoltage4,
                  float& dacVoltage5, float& dacVoltage6, float& dacVoltage7, float& dacVoltage8,
                  float& dacVoltage9, float& dacVoltage10, float& dacVoltage11, float& dacVoltage12,
                  float& dacVoltage13, float& dacVoltage14, float& dacVoltage15, float& dacVoltage16,
                  float& dacVoltage17,
                  uint16_t& waveform0, uint16_t& waveform1, uint16_t& waveform2, uint16_t& waveform3,
                  uint16_t& waveform4, uint16_t& waveform5, uint16_t& waveform6, uint16_t& waveform7,
                  uint16_t& waveform8, uint16_t& waveform9, uint16_t& waveform10, uint16_t& waveform11,
                  uint16_t& waveform12, uint16_t& waveform13, uint16_t& waveform14);
```

Configures the information from the Fast CCD. Fills arguments with values depending on how the image/waveform data were taken.

```
int getFrameValue(FrameDetector det, int& frameWidth, int& frameHeight, unsigned short*& image );
```

Fetches the FCCD image data. Specify the detector (only SxrFccd is available as of 2012).

- PnCCD camera (used by the CAMP collaboration):

```
int getPnCcdValue (int deviceId, unsigned char*& image, int& width, int& height );
```

This camera has 4 links, each link provides a 512 x 512 x 16 bit image. This function combines the four images to a single 1024 x 1024 x 16 bit image.
`deviceId` can be PnCcd0 or PnCcd1, width and height are the number of pixels in each direction.

- Princeton camera:
  To get the image data (array of unsigned short), use `getPrincetonValue`, and to get other information, like the image size, camera exposure, temperature etc, use `getPrincetonConfig` and `getPrincetonTemperature`:

```
int getPrincetonConfig(Pds::DetInfo::Detector det, int iDevId,
                       int& width, int& height,  // image width and height in pixels
                       int& orgX, int& orgY,     // 0,0
                       int& binX, int&binY);     // 1,1

int getPrincetonValue(Pds::DetInfo::Detector det, int iDevId,
                      unsigned short *& image);  // pointer to first pixel element

int getPrincetonTemperature(Pds::DetInfo::Detector det, int iDevId,
                            float& temperature);
```

  fetches the configuration and data from the camera. `getPrincetonTemperature` is there to check the temperature of the camera at the time of data taking (not necessarily available for every shot).

Other functions:

Several other functions are available to get information from other monitors around the experimental setup. They may or may not be of importance to your data quality...

- Ipimb detector (Intensity Position, Intensity Monitor Board)

```
int getIpimbConfig(Pds::DetInfo::Detector det, int iDevId);
int getIpimbVolts(Pds::DetInfo::Detector det, int iDevId,
                  float &channel0, float &channel1, float &channel2, float &channel3);
```

  Measures intensity and position of the beam from scattered X-rays.

- Encoder detector

```
int getEncoderConfig   (Pds::DetInfo::Detector det, int iDevId);
int getEncoderCount(Pds::DetInfo::Detector det, int iDevId, unsigned int& encoderCount);
```

  Position of mirrors (SXR)

- DiodeFex (Diode feature extraction)

```
int getDiodeFexConfig (Pds::DetInfo::Detector det, int iDevId, float* base, float* scale);
int getDiodeFexValue (Pds::DetInfo::Detector det, int iDevId, float& value);
```

- Ipm detector Fex (Ipm feature extraction)

```
int getIpmFexConfig   (Pds::DetInfo::Detector det, int iDevId,
                       float* base0, float* scale0,
                       float* base1, float* scale1,
                       float* base2, float* scale2,
                       float* base3, float* scale3,
                       float& xscale, float& yscale);
int getIpmFexValue   (Pds::DetInfo::Detector det, int iDevId,
                      float* channels, float& sum, float& xpos, float& ypos);
```

- Front end enclosure Gas detector

```
int getFeeGasDet  (double* shotEnergy);
```

Gives you the shot energy to the array `shotEnergy[4]`.

- Electron beam monitor

```
int getEBeam(double& charge, double& energy, double& posx, double& posy,
             double& angx, double& angy);
int getEBeam(double& charge, double& energy, double& posx, double& posy,
             double& angx, double& angy, double& pkcurr);
```

Gives electron beam values for each of these doubles. The measured charge of the beam (in nC),
the measured energy of the beam (in MeV), the 2D position of the beam (in mm) away from the origin
(nominal beam position), and 2D angular position (in mrad) off the assumed direction. and the
pkcurr = current? in (Amps)

- Phase cavity montior

```
int getPhaseCavity(double& fitTime1, double& fitTime2, double& charge1,  double& charge2);
```

Gives you the phase cavity fit time (low and high?) and charges (before and after?).

- Event Receiver counter

```
int getEvrDataNumber()
```

number of Fifo pulses associated with this shot (usually 1 or 2). For each of these there is EvrData:

- Event Receiver data

```
 int getEvrData(int id, unsigned int& eventCode, unsigned int& fiducial, unsigned int& timeStamp );
```

eventCode tells you something about the beam quality of this pulse. Usually the event code is 140, meaning electrons were produced upstream
(beam was on). It does not tell you about the photon status.
fiducial is the higher timestamp of the pulse (end time)
timeStamp is the lower timestamp of the pulse (start time)

- EPICS values (Process variables)
Get integers, floats, strings from any EPICS channel (PV = process variable)

```
int getPvInt      (const char* pvName, int& value);
int getPvFloat    (const char* pvName, float& value);
int getPvString   (const char* pvName, char*& value);
```

Further analysis help

CSPad analysis page