Outdated: The XTC to HDF5 Translator

<documentation/link to new procedure to follow>

Introduction

The primary purpose of the XTC to HDF5 translation software is to provide a thorough translation of the data recorded in the XTC files of a LCLS experiment to the more general scientific format HDF5. Users can then analyze their data using any tool that supports HDF5, at SLAC or offsite. Important points to make about the Translator

1. By default, the Translator will apply calibration corrections to cspad and cspad2x2 detectors, but not epix or more recent detectors.
2. Options exist for users to skip calibrations and include the raw detector data.
3. Options exist to filter what detectors are translated
4. Advanced options exist for users to run their own code to produce output to be translated

However, regarding 4, new users interested in adding their own code into the translator are encouraged to look at other options. This is because the Translator is part of the psana module framework which is being deprecated – new users will have to learn how to write psana modules to use the Translator in this fashion. The most general alternative is to 4) is to develop your own code based on the Hdf5 building block examples that start here: Saving User Data in Hdf5. Simpler would be to see if the analysis that is computed in the littleData software developed by XPP apply to your experiment. XPP maintains a wiki: https://sites.google.com/site/xppwiki/ that includes information about the little data format, specifically here: https://sites.google.com/site/xppwiki/data-analysis/little-data, but best is to contact the experiment POC to see if the littleData software is applicable to your experiment.

These alternatives allow for a very simple schema that users can design themselves, as well as leveraging the analysis that XPP has developed, assuming it applies to your experiment. The translator is recommended for users that want a complete copy of the experiment data, but in the Hdf5 format, as well as returning users that want to leverage code developed for the Translator in the psana module framework.

The Translator should generally be run using the automatic hdf5 translation feature of the Web Portal of Experiments. Each experiment has a "HDF5 Translation" tab that provides this interface. Below we discuss features of the Translator for customization of the translation - filtering unwanted events or detectors as well as adding processed data. The web portal provides a mechanism to customize the translation using these features. Experiment POC's should be able to help with using these features.

A discussion of the hdf5 output format for translator can be found here:

The contents of HDF5 files - the output format. Users that will work with hdf5 output should review this document. Some key points are:

- Datasets need not be aligned. That is the 5th image in a detector dataset may come from a different event than the 5th record in a gas detector dataset. One can match up records from different datasets by use the time datasets.
- One should use the _mask datasets to identify valid data. A _mask dataset record is 1 when the corresponding record of the data dataset if valid, 0 if it is not. When the _mask record is 0, the data record will be all zeros and should not be processed. The mask is 0 when the xtc data is

damaged. The type damaged data can then be found in the _damage dataset. The main reason to record blanks in the data datasets when damage occurs is to keep datasets as aligned as possible.
- The hdf5 group hierarchy has the following levels: run, calib cycle. type, source - regular event data is organized into datasets that live at the source level. Epics is special, rather then the two groups type and source, there are three groups for epics: type, source and epics pv name. Epics aliases live alongside epics pv names in this group hierarchy. Finally configuration data (that usually arrives once) is found in subgroups to the configure groups at the top of the hdf5 hierarchy.

Running the Translator

Generally, users should use the HDF5 Translation tab for their experiment on the web portal of experiments. This allows users to automatically translate runs as they become available and are in progress. In addition, Translation can produce large amounts of data that needs to be managed by LCLS. The HDF5 Translation page includes two options for translation - standard and monitoring. Most users will use standard. The differences are, when run through the web portal:

- Standard translator:
  - directly reads the large xtc files
  - does not start until the xtc files start to appear on the offline file system
  - translation jobs run in the psana queue (fast network link to read)
  - output is to the ana file system (fast network link to write)
  - output is one Hdf5 file per run in the experiments hdf5 folder
  - the hdf5 files are added to the LCLS file manager which means they are archived to tape and can be restored if they were purged from disk.
- Monitoring translator:
  - reads the small xtc files (.smd.xtc) and the large
  - starts as soon as xtc files appear on the fast feedback nodes
  - translation jobs run in the psnehq or psfehq, or priority queues (fast network link to read from ffb)
  - output is to the ana file system (slower network link to write)
  - output is a master file, and one hdf5 file per calib cycle (step)
  - output is written to the experiments scratch folder. It is not backed up and is eligible for purging according to the Data Retention Policy Version 2.

The monitoring translator is intended for Multi-calib cycle experiments. It uses MPI to translate different calib cycles in parallel. Typically users will filter the large detectors and write smaller summary information in order to seed up translation for monitoring purposes.

Calibrations

The default behavior for the Translator is to calibrate just cspad2x2 or Full cspad. To obtain the calibrations for other area detectors, as well as to use more recent calibration code, see Translating the Detector Interface.

Running by Hand

You can also run the translator by hand. You run it as you would any other psana module. Either through psana command line options or by writing a psana configuration file. The module is Translator.H5Output. When using the psana command line interface to run the module, the only option that is required to give to the Translator is the name of the output file. This must be a fully qualified filename, with the output directory. For example:

```
psana -m Translator.H5Output -o Translator.H5Output.output_file=exp-run001.h5 exp=xpptut13:run=71
```

would invoke the translator. It will translate all the xtc files in run 71 of the xpptut13 dataset. This runs with default values for all the translator options. These are the recommended option values to use for translation. The options include gzip compression at level 1 and no filtering on events or data. The Translator does not overwrite an existing h5 output file by default (set the option overwrite=true to overwrite the output). There are many different options you can set for translation that are discussed below. The easiest way to try different translator options is to write a psana.cfg file. Later in this document, we include a long psana config file that includes extensive documentation on all the translator options. It is recommended that users copy and modify this document to try different options.

When running the Translator directly, you can make use of the same parallel compression library that the automatic translation system uses by setting the following environment variables (below is for bash):

```
export PAZLIB_MAX_THREADS=8
export LD_PRELOAD=libpazlib.so
```

before running. Automatic translation also calibrates all cspad data for which calibration constants have been deployed. To achieve this by hand, one can load the module cspad_mod.CsPadCalib as well (it must be loaded before the Translator). When running this module, it is useful to get some of its diagnostic messages to verify that it has found calibration files. After setting PAZLIB_MAX_THREADS and LD_PRELOAD, one could additionally set

```
export MSGLOGCONFIG=CalibDataProxy=trace
```

before running

```
psana -m cspad_mod.CsPadCalib,Translator.H5Output -o Translator.H5Output.output_file=exp-run001.h5 exp=xpptut13:
run=71
```

to translate in the same manner as the automatic translation system. Better I/O is achieved by writing output to the scratch folder of the experiment rather than a users home directory (on the LCSL system) and running translation through the batch system using the appropriate queue.

Split Scan Mode / Monitoring Translator

The Translator supports split scan mode. In this mode, calib cycles are written to separate hdf5 files. A master file will have external links to these separate hdf5 files. Users need only work with the master file. The master file uses the same schema as one finds without split scan mode. Little modification to user code is required when working with the master file. What is required, is following external links (see below for tips on this). Not all experiments use more than one calib cycle. For experiments that use one calib cycle per run, split scan mode provides no benefit. The main reason users will opt for the split scan translator it to achieve the fastest possible translation for the purposes of online monitoring. Often, this involves custom configuration to reduce the DAQ data translated, as well as adding results from users own code to the translation. The split scan translator is implemented using MPI and translates different calib cycles in parallel. It has its own driver program to be launched using MPI called h5-mpi-translate. For this reason, the split scan translator is also referred to as the monitoring translator.

The split scan translator has a master/worker architecture. The single master process reads through the data and finds where the calib cycles start. It then assigns calib cycles to the worker pool.

Running Split Scan

The simplest way to launch the split scan translator is through the web portal of experiments - the monitoring choice on the HDF5 Translation tab. Before covering the options available through the web page, we'll look at launching jobs by hand. Here is an example command for launching the mpi based splitscan Translator for data that has already finished being written to the offline file system:

```
bsub -q psanaq -n 9 -o translate_%J.out mpirun h5-mpi-translate -m cspad_mod.CsPadCalib,Translator.H5Output -o
Translator.H5Output.output_file=mydir/split.h5 exp=xpptut13:run=71:smd
```

Note the use of :smd for the dataset. This allows the master process of h5-mpi-translate to index the calib cycles quickly. When running h5-mpi-translate by hand, forgetting to add :smd will greatly degrade performance.

The first few arguments for bsub set up the batch job, everything after h5-mpi-translate are arguments for the mpi split scan translator. For the purposes of online monitoring, we often want to translate while taking data, and we will want to use the appropriate high priority queue for the instrument our experiment uses. These queues are psnehprior and psfehprior - for instruments in the near hall and far hall respectively. In addition, there are certain arguments for launching the batch job that seem to increase translation speed - at the expense of using a large number of resources in the queue. An example launch command for an XPP experiment in the near hall might look like

```
bsub -q psnehprior -n 9 -x -R "span[ptile=2]" -o translate_%J.out mpirun h5-mpi-translate -m cspad_mod.CsPadCalib,
Translator.H5Output -o Translator.H5Output.output_file=mydir/split.h5 exp=xpp7815:run=189:smd:live:dir=/reg/d/ffb
/xpp/xppf7815/xtc
```

Note - it is important to make sure that the priority queues are available for experiments during their scheduled time. In particular, if one is doing online monitoring for an experiment running during the day shift using a high priority queue, one must stop using the queue if it will be used by an experiment running during the night shift (use the "stop all" button if translating through the web portal of experiments). At that point one would switch to doing translation using the offline psanaq, and the offline file system rather than the ffb. Optimal I/O performance requires coordinating which queue you use with where the data is read from, and written to. In particular, the ffb data location should only be used for the high priority queues. See Batch Nodes And Queues for more details on the different queues, In terms of where to write data, write to an experiment folder like ftc or scratch - don't write to your home directly.

batch arguments

The bsub batch arguments used on the priority queue are

- **-x** no other batch jobs run on the nodes used for this job
- **-R "span[ptile=2]"** only run two processes one each node.
- **-n 9** use 9 processes for the job (1 will be the master, and 8 for calib cycles, up to 8 calib cycles will be translated simultaneously). The master process is always the last process. By using 9 processes and ptile=2, and -x, the master process runs by itself on one node. No workers will be doing I/O on the node. This gives the master the most possible resources for its job of finding calib cycles. In general, for best performance, if n is the number of processes and k the processes per node, (the ptile value) choose them so that n mod k == 1
- The job output file (translate_39283.out, where 39283 will be whatever job number the batch system assigns) will record timing information for the master and workers.

These arguments dedicate significant resources of the priority queue to the translation of each run to achieve the fastest possible translation. There are two costs to this approach. The first is less resources for other jobs in the queue, like translating other runs. The second is increasing the likelihood of failure by using a problematic node. If one of the nodes in the queue is having trouble, these options increase our odds of running a job on it. In general we do not recommend these arguments for the offline queue: psanaq.

The translator, when compressing, spends about 70% of its time as a multi-threaded application (using 9 of the 12 cores on the nodes) - thus the ptile=2. Even when not compression, using lower ptile values (1,2,3) seems to increase overall performance. I believe this is due to the I/O intensive nature of the Translator. On paper, we expect our filesystem and network links to perform just as well with ptile=12 (the default) as opposed to ptile = 1,2,3, however in practice this is not what I've seen.

How many processes to use?

Presently, the split scan translator does not dynamically request new processes as it needs them, it uses whatever pool it has been launched with (the -n batch argument above). Consequently the user needs to pick the optimal number of processes to use. Ideally, we want a worker to be free when each new calib cycle is discovered. Assuming the fast_index option is used (see below), and optimal batch arguments like -x, ptile=2 with an odd number of processes are used, the master should keep up with live data at 120hz. Then it is a question of how slow the calib cycle translation is. For 1 full cspad with compression, I would allow for 11hz to translate calib cycles. 120/11 means I want 11 workers, so n=12, and I'll make it n=13 to get the master running by itself.

The Translator output, captured in log files by the bsub -o option, provide useful information on the translation rate for the workers, the rate the master gets through the data, and how many calib cycles each worker translated (if you see workers that didn't translate any calib cycles, you have assigned more processes to the translation job then the system can use).

Translator arguments

Looking again at the bsub command line:

```
bsub -q psanaq -n 9 -o translate_%J.out mpirun h5-mpi-translate -m cspad_mod.CsPadCalib,Translator.H5Output -o
Translator.H5Output.output_file=mydir/split.h5 exp=xpptut13:run=71
```

You will see that h5-mpi-translate, rather than psana, is the driver program for mpi based split scan translation. h5-mpi-translate takes the same arguments as psana does. In particular, h5-mpi-translate understands the -c file.cfg option so that options may be specified in a configuration file.

Operation

When h5-mpi-translate runs, one node (with the master process) will read through the data, identify the start of calib cycles, and communicate via MPI with the other 8 nodes. The other 8 nodes will do all the translation - telling the master when they are done and ready for a new job. The master will add links to the master h5 output file only after the separate calib files are done.

Generally, there will be one calib cycle file for each calib cycle. However to prevent too many calib cycle files from being produced for experiments that have only a few events per calib cycle, small calib cycles are grouped into one file. They are grouped until the total number of events in the file exceeds a threshold. The threshold defaults to around 100 but is configurable.

For the above command, assuming there are multiple calib cycles in run 71 of xpptut13 that each have at least 100 events, and that the batch job was 39283, you will produce the following files:

```
translate_39283.out
mydir/split.h5
mydir/split_cc0000.h5
mydir/split_cc0001.h5
...
```

When moving the hdf5 files, make sure they all reside in the same directory. The links from the master to the calib cycle files assume they are in the same directory.

Calib files in sub directory

While the split scan translator defaults to putting all files, the master file, and translated calib cycle files, into the same directory, users can optionally put the calib files into a sub-directory. The option is "split_cc_in_subdir". For example, when translating the above datasource, if one were to use a config file and add an option like so

```
[Translator.H5Output]
split_cc_in_subdir=True
```

Then one will get all the calib files in a directory called xpptut13_r0071_ccfiles and the translated files there:

```
mydir/xpptut13-r0071.h5
mydir/xpptut13-r0071_ccfiles/xpptut13-r0071_cc0000.h5
mydir/xpptut13-r0071_ccfiles/xpptut13-r0071_cc0001.h5
...
```

Live Data and Fast Indexing

For online analysis with live data, one of the impediments to keeping up with the data is the time it takes h5-mpi-translate's to read through the data to find the calib cycles. As long as the user specifies :smd in the dataset (which the web portal does) this should not be a problem. A deprecated option, fast_index, was used before we had small data. Using that option now generates an error, however if for some reason there are problems with the small data - users can fall back on the fast_index option by specifying fast_index_force=1. Below we document this deprecated option.

Typically the data is recorded in 6 or more separate files and if the small data files are not available, each must be read through to identify the start of calib cycles. Unfortunately this read speed can be 30-40hz for a typical experiment - far short of the 120hz we'd like to obtain. The deprecated fast_index_force option in h5-mpi-translate takes advantage of the unique signature of each new calib cycle, combined with the regular structure of the separate data files in order to limit the reading to just one of the files. In this way, the h5-mpi-translate master rank need only get through the large xtc data it reads/searches at 20hz to keep up with the data. Part of why fast_index is deprecated in favor of small data is that it is not guaranteed to work - in particular high level of damaged data degrades the regular structure of the DAQ files. This in turn will increase the fastindex search time to the point where it is no longer useful, or fails.

```
fast_index_force =1                 # do the fast indexing, be default it is off
fi_mb_half_block=12          # when fast indexing is on, use 12MB on each side, or 24MB for each block that is
searched
fi_num_blocks=40             # this it half the number of 'other' blocks to try. The translator will try 1 +
2*40 = 81 blocks if this is 40 (about 1GB total search)
```

To obtain More information can be found in the Psana Configuration File and All the Options section below.

Reading While Translating

HDF5 presently has little support for reading a file that is being created, and in general does not recommend this. However the master file is written in a way to support this as well as possible. When using the MPI split scan translator, links are not added to the master file until after the calib files are done. Thus it is always safe to traverse those links. To see updates in the master file, users may need to shut down programs like Matlab and h5py and restart them. That is it may not be sufficient to close and reopen the master file within a Python or Matlab session.

EndData and Split Translation

The EndData feature, discussed below, provides a way for Psana user modules to translate data during beginRun and endRun. However in split scan mode, each calib cycle is translated by a separate translator, and each separate translator will create an independent instance of any Psana modules that have been specified in the h5-mpi-translate command line. Consequently if a Psana module outputs summary information during endRun, it will not be summary information for the whole run, just those calib cycles translated by the Translator that loaded it. Moreover the master file will make a link to the first EndData within Run:0000 that it finds. That is, if there are 10 different external calib cycle files, with 10 different EndRun groups, there will be a link to only one of them from the master file.

Working with External Links

When working with the master file, it is necessary to follow external links. For instance, to get a recursive listing of all the groups in the output file using h5ls, one must do

h5ls -r -E master.h5

as opposed to h5ls -r file.h5. The -E option instructs h5ls to follow external links. Similar functionality should exist in h5py, Matlab, and other software that works with HDF5.

Presentations

Attached is a link to a presentation given during the LCLS 2014 users meeting. It goves over dataset alignment as well as new features: Using_the_HDF5_Translator.pdf

Advanced Features

- filter out whole events from translation
- filter out certain data, by data type, or by data source, or key string
- write ndarray's that other modules add to the event store or configStore
- write std::string's that other C++ modules add to the event store or configStore
- include summary data for calib cycles

Calibrated Data

Calibration is handled by external psana modules. These modules will produce calibrated data and the translator will find it and translate it to the hdf5 file. Understanding this flow of data is not necessary for automatic translation, however if users want to customize calibration, some understanding of how psana modules pass data through the event store, and are configured through config files is necessary. The calibrated data will be distinguished from uncalibrated data with the use of a key in the event store. The key defaults to the value 'calibrated' but this is configurable through the psana.cfg file, in the section for the calibration module used. the translator provides special treatment for the calibration key. For the translator, the default value for the calibration key is 'calibrated' as well, but again, this is configurable through the psana.cfg file, in the section for Translator.H5Output. If the translator sees data with the key calibrated - it defaults to only translate data with the calibrated key and not the raw data. In the hdf5 file, one will find calibrated data where one would have otherwise found uncalibrated data. The calibrated key is not present in the hdf5 path names. This is different than what one finds for keys with ndarrays. For ndarrays the key is part of the h5 path name (see below). The the translator option skip_calibrated can be set to true to get the uncalibrated data instead of calibrated data.

Calibration makes use of calibration constants - such as pedestals and pixel status. A key difference between the translator and o2o-translate is where these calibration constants are found, and the datatypes used to store them. For the translator they are found in the group CalibStore to the current configure group. For example, if we translate the first event in a run of the cxi tutorial data where we add the cspad calibration module before the the translator module:

```
psana -n 1 -m cspad_mod.CsPadCalib,Translator.H5Output -o Translator.H5Output.output_file=calib.h5 exp=xpptut13:
run=71
```

Then we will see

```
h5ls -r calib.h5 | grep -i "calibstore\|cspad"  # this command will include the following output

/Configure:0000/Run:0000/CalibCycle:0000/CsPad2x2::ElementV1/XppGon.0:Cspad2x2.0/common_mode Dataset {1/Inf}
/Configure:0000/Run:0000/CalibCycle:0000/CsPad2x2::ElementV1/XppGon.0:Cspad2x2.0/data Dataset {1/Inf}
/Configure:0000/Run:0000/CalibCycle:0000/CsPad2x2::ElementV1/XppGon.0:Cspad2x2.0/element Dataset {1/Inf}
...
/Configure:0000/Run:0000/CalibCycle:0000/CsPad2x2::ElementV1/XppGon.0:Cspad2x2.1/common_mode Dataset {1/Inf}
/Configure:0000/Run:0000/CalibCycle:0000/CsPad2x2::ElementV1/XppGon.0:Cspad2x2.1/data Dataset {1/Inf}
/Configure:0000/Run:0000/CalibCycle:0000/CsPad2x2::ElementV1/XppGon.0:Cspad2x2.1/element Dataset {1/Inf}
...
/Configure:0000/CalibStore/pdscalibdata::CsPad2x2PedestalsV1/XppGon.0:Cspad2x2.0/pedestals Dataset {185, 388, 2}
/Configure:0000/CalibStore/pdscalibdata::CsPad2x2PedestalsV1/XppGon.0:Cspad2x2.1/pedestals Dataset {185, 388, 2}
/Configure:0000/CalibStore/pdscalibdata::CsPad2x2PixelStatusV1/XppGon.0:Cspad2x2.0/status Dataset {185, 388, 2}
/Configure:0000/CalibStore/pdscalibdata::CsPadCommonModeSubV1/XppGon.0:Cspad2x2.0/data Dataset {SCALAR}
...
```

Things to note:

- There are common_mode datasets included with the data
- Both cspad sources have a pedestal dataset in CalibStore
- Only XppGon.0:Cspad2x2.0 has a common mode dataset in the calibstore.

When one looks at the common_mode dataset for XppGon.0:Cspad2x2.1, one sees the values are -10000, indicating no common mode calibration was done. Documentation on the CsPadCalib module is found in psana - Module Catalog.

An issue users may run into is understanding what calibration was done and recovering the raw data just from examining the hdf5 output. In the case of cspad, an understanding of the CsPadCalib module along with the what is in the hdf5 file does allow one to recover the uncalibrated data. This may not be possible with other calibration modules and detectors, in particular if nonlinear calibration algorithms are applied, such as applying a threshold.

Filtering Calibrated Data

By default, when the Translator sees both the original xtc data and a calibrated version of it, it writes the calibrated data in place of the xtc data. For automatic translation, we always load the module that will calibrate cspad if possible. If users do not want the calibrated data but prefer the raw data, they can set the option

skip_calibrated = true

If the user wants neither the calibrated nor the raw cspad, they they should use type filters, that is setting

Cspad = exclude
Cspad2x2 = exclude

There will be no cspad in the translation (including the cspad configuration data as well as event data). This can be useful if one is using other modules to produce ndarrays from the calibrated data and one only wants the final processed ndarrays in the translation.

PNCCD::FullFrame

This data is no longer translated. FullFrame is a copy of Frames with a more convenient interface for psana users, it is not considered to be as useful for hdf5 files. User's interested in having FullFrame written into their hdf5 files rather than the original Frames data should make a feature request.

Filtering Events

Since the translator runs as a psana module, it is possible to filter translated events through psana options and other modules. psana options allow you to start at a certain event, and process a certain number of events. Moreover a user module that is loaded before the Translator module can tell psana that it should not pass this event on to any other modules, hence the Translator.H5Output module will never see the event and it will not get translated.

One can also filter events by putting an object in the event store with a special key string. To use this mechanism, a module must put an object in the eventStore with a key that starts with

```
do_not_translate
```

For example, if a C++ module implements the event method to do the following:

filtering

```
  virtual void event(Event& evt, Env& env) {
        boost::shared_ptr<int> dummyVariable = boost::make_shared<int>();
    evt.put(dummyVariable,"do_not_translate");
  }
```

Then none of the event data will get translated in any of the calib cycles.

Filtering from Python Modules

A Python module can use standard psana features to skip events as discussed above. It can also add any Python object into the event store that has the key "do_not_translate".

Below is a complete example. First we make a release environment, and create a package for our Python Module that will filter events.

Suppose we want to filter based on the photon count of the calibrated cspad from the CxiDs1.0:Cspad.0 source in the tutorial data. Rather then work with the quad's of the cspad, we will use an image producer module so that we can work with a 2D image. Further documentation on the various calibration modules, including image producers, can be found at psana - Module Catalog. We now add the following two files:

myrel/trans.cfg

```
[psana]
modules = cspad_mod.CsPadCalib \
          CSPadPixCoords.CSPadImageProducer \
          mypkg.mymod \
          Translator.H5Output
events = 10
files = exp=cxitut13:run=1150

[CSPadPixCoords.CSPadImageProducer]
source        = DetInfo(CxiDs1.0:Cspad.0)
key           = calibrated
imgkey        = image

[Translator.H5Output]
deflate=-1
shuffle=False
overwrite=True
output_file=cxitut13-run1150-filt.h5
```

and the file

myrel/mypkg/src/mymod.py

```
import psana

class mymod(object):
    def __init__(self):
        self.threshold = self.configInt('threshold',176000000)
        self.source = self.configSrc('source','DetInfo(CxiDs1.0:Cspad.0)')

    def event(self, evt, env):
        image = evt.get(psana.ndarray_int16_2,
                        self.source,
                        'image')
            if image is None: return
        photonCount = image[:].sum()
        if photonCount < self.threshold:
            self.skip()
```

After putting these files in place, and doing

```
scons
```

we can run this example by

```
psana -c trans.cfg
```

The configuration file trans.cfg sets up a module chain with 4 modules: cspad_mod.CsPadCalib, CSPadPixCoords.CSPadImageProducer, mypkg.mymod, Translator.H5Output. The first module calibrates all cspad it finds. The second module turns a cspad from a specific source into an image - placing quads and ascics in the correct position based on geometry. After these two modules, we load our own module - mypkg.mymod. Finally the Translator module runs last.

Reading through trans.cfg you will see how the raw cspad moves through the event store. The default behavior of cspad_mod.CsPadCalib is to place calibrated cspad in the Event with the key "calibrated". The CSPadPixCoords.CSPadImageProducer module has been told to look for the "calibrated" input key, for the source DetInfo(CxiDs1.0:Cspad.0) and produce an image with the key "image".

In mymod.py, we see a class called mymod derived from object. It is important that the class name be the same as the file name. This is part of how psana finds the class. In the event, the module gets data of type psana.ndarray_int16_2. Identifying the correct type to use can be a challenge. Starting with code in event() that does

```
print evt.keys()
```

shows what the keys look like. After getting a valid image, a sum and simple threshold is performed.

Note the option

events = 10

in the psana section of the config file. This means one would translate 10 events in the data. This is just for testing and development. One would remove the option, or set it to 0 for a full translation. With events=10, after translation, if one does

```
h5ls -r cxitut13-run1150-filt.h5  | grep -i "ndarray"
/Configure:0000/Run:0000/CalibCycle:0000/ndarray_const_int16_2/CxiDs1.0:Cspad.0__image/data Dataset {5/Inf}
```

one sees that only 5 events were translated. The other 5 were skipped.

Another point to make about this example is that the cspad is effectively getting translated twice. The Translator is going to see the event keys:

EventKey(type=psana.CsPad.DataV2, src='DetInfo(CxiDs1.0:Cspad.0)')
EventKey(type=psana.CsPad.DataV2, src='DetInfo(CxiDs1.0:Cspad.0)', key='calibrated')
EventKey(type=psana.ndarray_int16_2, src='DetInfo(CxiDs1.0:Cspad.0)', key='image')

The Translator's default behavior is to treat the key 'calibrated' as special. Since the first two keys differ only by the keystring 'calibrated', the Translator assumed the one marked 'calibrated' should replace the first in the translation. Hence it will not translate the raw cspad. It only translated the calibrated cspad. However the Translator does not know that the ndarray with key 'image' is a copy of the cspad. If one is only going to work with the 'image' array data and not the 'calibrated' cspad data, one could add the filtering option

Cspad=exclude

To the Translator.H5Output section of the config file. Then none of the cspad data will be translated (including both the configuration cspad object as well as event data) while the 'image' arrays will still be translated.

Filtering Types

The psana.cfg file accepts a number of parameters that will filter out sets of psana types. For example setting

```
EBeam = exclude
```

would cause any of the types Psana::Bld::BldDataEBeamV0, Psana::Bld::BldDataEBeamV1, Psana::Bld::BldDataEBeamV2, Psana::Bld::BldDataEBeamV3 or Psana::Bld::BldDataEBeamV4 to be excluded from translation.

All types are translated by default. To exclude a few types, you can add lines like EBeam = exclude to the psana.cfg file. You can also list them with the type_filter parameter:

```
type_filter exclude EBeam Andor
```

The type_filter parameter is useful for including a few types:

```
type_filter include CsPad Frame
```

A shortcut is available to turn off translation of all the Xtc data:

```
type_filter exclude psana
```

One would use this to only translate user module data, such as ndarrays, strings and newly registered types.

The complete list of type aliases that users can use to filter is found in the default_psana.cfg file included below.

Src Filtering

Specific src's can be filtered by providing a list such as

```
src_filter = exclude NoDetector.0:Evr.2  CxiDs1.0:Cspad.0  CxiSc2.0:Cspad2x2.1  EBeam  FEEGasDetEnergy  CxiDg2_Pim
```

the syntax for a src in the filter list is what is supported by the Psana::Source class. This is a flexible syntax allowing for several ways to specify a src. It will match any detector or device number if this is not specified. See the section Psana Configuration File and all Options below for more details. If DAQ src aliases are present in the xtc file, these can be used for src filtering as well. For example if the alias

```
acq01 -> SxrEndstation.0:Acqiris.0
```

is present, one can do

```
src_filter = exclude acq01
```

to exclude all data from the SxrEndstation.0:Acqiris.0 src.

Type/Src (EventKey) filtering

Sometimes the type and src filtering do not offer enough control over what to filter. One can also (as of ana-0.15.1) filter based on individual event keys - that is specify a type and a src together (and optionally a key string, but this is usually not necessary). For example, suppose when doing EventKeys on a run in your experiment, one has the following pieces of data in the event:

```
  EventKey(type=Psana::Camera::FrameV1, src=DetInfo(XppSb4Pim.1:Tm6740.1), alias="yag3")
  EventKey(type=Psana::Package::ProjV1, src=DetInfo(XppSb4Pim.1:Tm6740.1), alias="yag3")
  EventKey(type=Psana::Camera::FrameV1, src=DetInfo(XppEndstation.0:Opal1000.0), alias="opal_0")
  EventKey(type=Psana::Camera::FrameV1, src=DetInfo(XrayTransportDiagnostic.0:Opal1000.0), alias="xtcav")
```

and one doesn't want to translate the Camera::FrameV1 from yag3, but one does want to translate Package::ProjV1. Perhaps ProjV1 is a projection of the large detector, and this is all that is necessary for analysis. One also wants to translate the Camera::FrameV1's from the other two sources. This could be achieved by setting

```
  eventkey_filter = exclude Frame__yag3
```

That is, one can combine one of the Type Aliases from the type filter with a src, using a double underscore, __ , to separate the two. One can make the exclude list as long as one likes (or make it an include list). The Translator will fatally stop if a type alias or src is not understood, or if the __ is not found.

Writing User Data

The translator will write NDarrays, C++ std::strings, and C++ types that the user registers. Presently, registering new types is an advanced feature that requires familiarity with hdf5 programming. To add data to the translation, one must write a Psana module that adds this data into the event or configStore before the Translator.H5Output module runs.

Event vs. ConfigStore, EndData subgroups

Most user modules will add data to the event. Such data will be written into *stacked* datasets, that is a 1D dataset of a type X based on the user data. In this 1D dataset, there will be one entry for each event the user module added data. Alongside this data, in a dataset named "data" will be a dataset named "time". The "time" dataset will have the event id's corresponding to the Psana Events the from which the user module added data.

Data added to the configStore is not written out in "stacked" datasets. It is written out in "one shot". The intention is that users may add some configuration during beginrun or begincalibcycle as well as some summary information during endcalibcycle or endrun. It is not recommended that users add data to the configStore for the purpose of translation during regular events. During all three of endcalibcycle, endrun and endjob, the Translator will check for new data in the configStore(). If it finds it, it will create a subgroup called EndData in the appropriate place. For example

```
/Configure:0000/Run:0000/CalibCycle:0000/EndData            # triggered by new data in the configStore() found
during endcalibcycle
/Configure:0000/Run:0000/EndData                            # triggered by new data in the configStore() found
during endrun
/Configure:0000/EndData                                     # triggered by new data in the configStore() found
during endjob
```

In this way, when a user Psana module processes endcalibcycle or endrun, it can add summary data to the configStore that will be picked up by the Translator. Psana modules could also add configuration information during beginrun or beginjob.

One limitation users may run into is overwriting keys - Psana does not allow Python modules to replace ndarrays in the configStore as C++ modules may be relying on the data to be unchanging. So for example, if a user module is going to create new summary information for each endcalibcycle, they must use different keys.

NDArrays and Strings

ndarrays (up to dimension 4 of the standard integral types, floats and doubles) as well as std::string's that are written into the event store will be written to the hdf5 by default. ndarrays can be passed to the Translator by Python modules as well as C++ modules. The schema for translating is to join the source and keystring with double underscore. For instance, given a psana user module that looks like this

```
import numpy as np
import psana

def MyModule(object):
  def __init__(self):
    self.src = psana.Source("DetInfo(XppEndstation.0:Opal1000.0)")

  def event(self, evt, env):
        a = np.zeros(3)
    evt.put(a,"mykeyA")
    evt.put("my string", "mykeyB")
    evt.put(a,self.src, "mykeyA")
    evt.put("my string", self.src, "mykeyB")
```

One would get these new groups in the HDF5 file:

```
/Configure:0000/Run:0000/CalibCycle:0000/ndarray_float64_1/noSrc__mykeyA/data
/Configure:0000/Run:0000/CalibCycle:0000/ndarray_float64_1/noSrc__mykeyA/time
/Configure:0000/Run:0000/CalibCycle:0000/ndarray_float64_1/XppEndstation.0:Opal1000.0__mykeyA/data
/Configure:0000/Run:0000/CalibCycle:0000/ndarray_float64_1/XppEndstation.0:Opal1000.0__mykeyA/time
/Configure:0000/Run:0000/CalibCycle:0000/std::string/noSrc__mykeyB/data
/Configure:0000/Run:0000/CalibCycle:0000/std::string/noSrc__mykeyB/time
/Configure:0000/Run:0000/CalibCycle:0000/std::string/XppEndstation.0:Opal1000.0__mykeyB/data
/Configure:0000/Run:0000/CalibCycle:0000/std::string/XppEndstation.0:Opal1000.0__mykeyB/time
```

Note that data put in the event store without a src specified go under a group that starts with "noSrc". All data gets a "stacked" dataset named data, and a time dataset.

Note, the type group name for ndarrays is fully qualified by the template arguments, some examples of type names are

```
ndarray_int8_1           # a one dimensional array of 8 bit signed integers    (the C type char)
ndarray_uint8_2          # a two dimensional array of 8 bit unsigned integers
ndarray_int32_1          # a one dimensional array of 32 bit signed integers   (the C type int)
ndarray_uint64_3         # a 3D array of 64 bit unsigned integers
ndarray_float32_2        # a 2D array of 32 bit floats  (the C type float)
ndarray_float64_1        # a 3D array of 64 bit floats (the C type double)
```

These names agree with what users find in the Python interface to psana.

Less common are the names used to store an ndarray of const data. An example name for such data is

```
ndarray_const_float32_2
```

Fixed Dimensions vs. Variable Dimensions

The Translator defaults to using a fixed set of dimensions for all the ndarrays that go into the same dataset. The array received for the first data of the dataset determine these dimensions. For example, if from python one did

```
event.put(numpy.zeros((3,4),"mykey")
```

during the first event, but then

```
event.put(numpy.zeros((5,4),"mykey")
```

during the second event, the Translator would throw an error. One option is to start a new dataset during the second event with a different key:

```
event.put(numpy.zeros((5,4),"mykey_larger")
```

The Translator supports variable length arrays in the same dataset, as long as the variation is only in the slow dimension. To use this option, one informs the Translator to set up a dataset of variable length arrays by pre-pending 'translate_vlen:' to the start of the keys. For example:

```
event.put(numpy.zeros((3,4),"translate_vlen:mykey")        # event one
event.put(numpy.zeros((5,4),"translate_vlen:mykey")        # event two
```

Now both ndarrays go to the same dataset, and the underlying hdf5 type changes to a vlen type of 1D arrays with dimension 4. The type name in the hdf5 path changes to indicate vlen, it will be

```
/ndarray_float32_2_vlen/noSrc__mykey
```

as opposed to

`/ndarray_float32_2/noSrc__mykey`

in the case of fixed length datasets.

Psana Configuration File and all Options

When running the translator as a psana module, if is often convenient to create a psana.cfg file. The Translator package include the file default_psana.cfg which is a psana configuration file that describes all the options possible, with extensive documentation as to what they mean. Below we include this file for reference. To use this file, one could it and modify it. However it is not necessary to take the whole file - every value set is set to the default value. One could simply use this as a reference for those options values that one wants to change.

psana-translate default_psana.cfg file - all options

```
######################################################################
[psana]
# MODULES: any modules that produce data to be translated need be loaded
# **BEFORE** Translator.H5Output (such as calibrated data or ndarray's)
# event data added by modules listed after Translator.H5Output is not translated.
modules = Translator.H5Output
files = **TODO: SPECIFY INPUT FILES OR DATA SOURCE HERE**
######################################################################
[Translator.H5Output]
# The only option you need to set for the Translator.H5Output module is
# output_file. All other options have default values (explained below).
# TODO: enter the full h5 output file name, including the output directory
output_file = output_directory/h5output.h5
# By default, the Translator will not overwrite the h5 file if it already exists
overwrite = false
# # # # # # # # # # # # # # # # # # # # # #
# EPICS FILTERING
# The Translator can store epics pv's in one of three ways, or not at all.
# set store_epics below, to one of the following:
#
# updates_only   stores an EPICS pv when it has been updated in the psana epics store.
#                For xtc input this happens whenever EPICS data is present in a datagram.
#                Note - many EPICS pvs are not present in every shot. A dataset
#                for an EPIC pv is often shorter than the total number of events.
#                Experiments with many short calib cycles may have some calib cycles where
#                no EPICS pv's show up. Users would then have to look back through several
#                calib cycle's to find the latest value of a pv.
#
# calib_repeat   This is the same as updates_only except that each calib cycle starts with
#                the most recent value of each pv. This makes it easier to find pv's in a
#                calib cycle. For experiments with many short calib cycles, it can produce
#                more datasets than neccessary.
#
# always         For each event, store the most recent value of the EPICs pv. Produces
#                longer datasets than neccessary, but makes it easier to find the latest
#                pv for an event.
#
# no             epics pv's will not be stored. You may also want to set Epics=exclude
#                (see below) if you do not want the epics configuration data stored
# The default is 'calib_repeat'
store_epics = calib_repeat
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# FILTERING
#
# By default, all xtc data is Translated and many ndarrays that user modules (if any)
# add are translated. The Translator can be configured to filter data based on
# a number of criteria. There are five options for filtering data:
#
#    type filtering   -  for example, exclude all cspad, regardless of the detector source
#    source filtering -  for example, exclude any data from a given detector source
#    key filtering    -  for example, include only ndarrays with a given key string
#    calibration      -  do not translate original xtc if a calibrated version is found
#    eventkey filtering -very fine control over filtering - specify type,src and key
```

```
#
# Type filtering is based on sets of Psana data types. If you know what detectors or
# devices to filter, leave type filtering alone and go to src_filter.
#
# Type filtering has the highest precedence, then key filtering, then source
# filtering, then "full" event key filtering, and lastly calibration filtering. When the
# Translator sees new data, it first checks the type filter. If it is a filtered type
# (or unknown type) no further translation occurs with the data - regardless of src or key.
# For data that gets past the type filter, the Translator looks at the src and key. The src
# filter is only applied to data that has an empty key string. To filter data with key strings,
# use the key filter, or full eventkey filter (if fine control is needed). Data with the special
# calibration key string are handled via the calibration filtering.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# TYPE FILTERING
#
# One can include or exclude a class of Psana types with the following
# options. Only the strings include or exclude are valid for these
# type filtering options.
#
# Note - Epics in the list below refers only to the epicsConfig data
# which is the epics alias list, not the epics pv's. To filter the epics pv's
# see the 'store_epics' option above.
AcqTdc = include               # Psana::Acqiris::TdcConfigV1, Psana::Acqiris::TdcDataV1
AcqWaveform = include          # Psana::Acqiris::ConfigV1, Psana::Acqiris::DataDescV1
Alias = include                # Psana::Alias::ConfigV1
AnalogInput = include          # Psana::Bld::BldDataAnalogInputV1
Andor = include                # Psana::Andor::ConfigV1, Psana::Andor::FrameV1
Andor3d = include              # Psana::Andor3d::ConfigV1, Psana::Andor3d::FrameV1
Arraychar = include            # Psana::Arraychar::DataV1
Control = include              # Psana::ControlData::ConfigV1, Psana::ControlData::ConfigV2, Psana::
ControlData::ConfigV3
Cspad = include                # Psana::CsPad::ConfigV1, Psana::CsPad::ConfigV2, Psana::CsPad::ConfigV3, Psana::
CsPad::ConfigV4, Psana::CsPad::ConfigV5, Psana::CsPad::DataV1, Psana::CsPad::DataV2
Cspad2x2 = include             # Psana::CsPad2x2::ConfigV1, Psana::CsPad2x2::ConfigV2, Psana::CsPad2x2::
ElementV1
DiodeFex = include             # Psana::Lusi::DiodeFexConfigV1, Psana::Lusi::DiodeFexConfigV2, Psana::Lusi::
DiodeFexV1
EBeam = include                # Psana::Bld::BldDataEBeamV0, Psana::Bld::BldDataEBeamV1, Psana::Bld::
BldDataEBeamV2, Psana::Bld::BldDataEBeamV3, Psana::Bld::BldDataEBeamV4, Psana::Bld::BldDataEBeamV5, Psana::Bld::
BldDataEBeamV6, Psana::Bld::BldDataEBeamV7
Encoder = include              # Psana::Encoder::ConfigV1, Psana::Encoder::ConfigV2, Psana::Encoder::DataV1,
Psana::Encoder::DataV2
Epics = include                # Psana::Epics::ConfigV1
Epix = include                 # Psana::Epix::ConfigV1, Psana::Epix::ElementV1, Psana::Epix::ElementV2, Psana::
Epix::ElementV3
Epix100a = include             # Psana::Epix::Config100aV1, Psana::Epix::Config100aV2
Epix10k = include              # Psana::Epix::Config10KV1
EpixSampler = include          # Psana::EpixSampler::ConfigV1, Psana::EpixSampler::ElementV1
Evr = include                  # Psana::EvrData::ConfigV1, Psana::EvrData::ConfigV2, Psana::EvrData::ConfigV3,
Psana::EvrData::ConfigV4, Psana::EvrData::ConfigV5, Psana::EvrData::ConfigV6, Psana::EvrData::ConfigV7, Psana::
EvrData::DataV3, Psana::EvrData::DataV4
EvrIO = include                # Psana::EvrData::IOConfigV1, Psana::EvrData::IOConfigV2
Evs = include                  # Psana::EvrData::SrcConfigV1
FEEGasDetEnergy = include      # Psana::Bld::BldDataFEEGasDetEnergy, Psana::Bld::BldDataFEEGasDetEnergyV1
Fccd = include                 # Psana::FCCD::FccdConfigV1, Psana::FCCD::FccdConfigV2
Fli = include                  # Psana::Fli::ConfigV1, Psana::Fli::FrameV1
Frame = include                # Psana::Camera::FrameV1
FrameFccd = include            # Psana::Camera::FrameFccdConfigV1
FrameFex = include             # Psana::Camera::FrameFexConfigV1
GMD = include                  # Psana::Bld::BldDataGMDV0, Psana::Bld::BldDataGMDV1, Psana::Bld::BldDataGMDV2
GenericPgp = include           # Psana::GenericPgp::ConfigV1
Gsc16ai = include              # Psana::Gsc16ai::ConfigV1, Psana::Gsc16ai::DataV1
Imp = include                  # Psana::Imp::ConfigV1, Psana::Imp::ElementV1
Ipimb = include                # Psana::Ipimb::ConfigV1, Psana::Ipimb::ConfigV2, Psana::Ipimb::DataV1, Psana::
Ipimb::DataV2
IpmFex = include               # Psana::Lusi::IpmFexConfigV1, Psana::Lusi::IpmFexConfigV2, Psana::Lusi::IpmFexV1
L3T = include                  # Psana::L3T::ConfigV1, Psana::L3T::DataV1, Psana::L3T::DataV2
OceanOptics = include          # Psana::OceanOptics::ConfigV1, Psana::OceanOptics::ConfigV2, Psana::
OceanOptics::DataV1, Psana::OceanOptics::DataV2, Psana::OceanOptics::DataV3
Opal1k = include               # Psana::Opal1k::ConfigV1
```

```
Orca = include                 # Psana::Orca::ConfigV1
Partition = include            # Psana::Partition::ConfigV1
PhaseCavity = include          # Psana::Bld::BldDataPhaseCavity
PimImage = include             # Psana::Lusi::PimImageConfigV1
Pimax = include                # Psana::Pimax::ConfigV1, Psana::Pimax::FrameV1
Princeton = include            # Psana::Princeton::ConfigV1, Psana::Princeton::ConfigV2, Psana::Princeton::
ConfigV3, Psana::Princeton::ConfigV4, Psana::Princeton::ConfigV5, Psana::Princeton::FrameV1, Psana::Princeton::
FrameV2
PrincetonInfo = include        # Psana::Princeton::InfoV1
Quartz = include               # Psana::Quartz::ConfigV1, Psana::Quartz::ConfigV2
Rayonix = include              # Psana::Rayonix::ConfigV1, Psana::Rayonix::ConfigV2
SharedAcqADC = include         # Psana::Bld::BldDataAcqADCV1
SharedIpimb = include          # Psana::Bld::BldDataIpimbV0, Psana::Bld::BldDataIpimbV1
SharedPim = include            # Psana::Bld::BldDataPimV1
Spectrometer = include         # Psana::Bld::BldDataSpectrometerV0, Psana::Bld::BldDataSpectrometerV1
TM6740 = include               # Psana::Pulnix::TM6740ConfigV1, Psana::Pulnix::TM6740ConfigV2
TimeTool = include             # Psana::TimeTool::ConfigV1, Psana::TimeTool::ConfigV2, Psana::TimeTool::DataV1,
Psana::TimeTool::DataV2
Timepix = include              # Psana::Timepix::ConfigV1, Psana::Timepix::ConfigV2, Psana::Timepix::ConfigV3,
Psana::Timepix::DataV1, Psana::Timepix::DataV2
TwoDGaussian = include         # Psana::Camera::TwoDGaussianV1
UsdUsb = include               # Psana::UsdUsb::ConfigV1, Psana::UsdUsb::DataV1
pnCCD = include                # Psana::PNCCD::ConfigV1, Psana::PNCCD::ConfigV2, Psana::PNCCD::FramesV1
# user types to translate from the event store
ndarray_types = include        # ndarray<int8_t,1>, ndarray<int8_t,2>, ndarray<int8_t,3>, ndarray<int8_t,4>,
ndarray<int16_t,1>, ndarray<int16_t,2>, ndarray<int16_t,3>, ndarray<int16_t,4>, ndarray<int32_t,1>,
ndarray<int32_t,2>, ndarray<int32_t,3>, ndarray<int32_t,4>, ndarray<int64_t,1>, ndarray<int64_t,2>,
ndarray<int64_t,3>, ndarray<int64_t,4>, ndarray<uint8_t,1>, ndarray<uint8_t,2>, ndarray<uint8_t,3>,
ndarray<uint8_t,4>, ndarray<uint16_t,1>, ndarray<uint16_t,2>, ndarray<uint16_t,3>, ndarray<uint16_t,4>,
ndarray<uint32_t,1>, ndarray<uint32_t,2>, ndarray<uint32_t,3>, ndarray<uint32_t,4>, ndarray<uint64_t,1>,
ndarray<uint64_t,2>, ndarray<uint64_t,3>, ndarray<uint64_t,4>, ndarray<float,1>, ndarray<float,2>,
ndarray<float,3>, ndarray<float,4>, ndarray<double,1>, ndarray<double,2>, ndarray<double,3>, ndarray<double,4>,
ndarray<const int8_t,1>, ndarray<const int8_t,2>, ndarray<const int8_t,3>, ndarray<const int8_t,4>,
ndarray<const int16_t,1>, ndarray<const int16_t,2>, ndarray<const int16_t,3>, ndarray<const int16_t,4>,
ndarray<const int32_t,1>, ndarray<const int32_t,2>, ndarray<const int32_t,3>, ndarray<const int32_t,4>,
ndarray<const int64_t,1>, ndarray<const int64_t,2>, ndarray<const int64_t,3>, ndarray<const int64_t,4>,
ndarray<const uint8_t,1>, ndarray<const uint8_t,2>, ndarray<const uint8_t,3>, ndarray<const uint8_t,4>,
ndarray<const uint16_t,1>, ndarray<const uint16_t,2>, ndarray<const uint16_t,3>, ndarray<const uint16_t,4>,
ndarray<const uint32_t,1>, ndarray<const uint32_t,2>, ndarray<const uint32_t,3>, ndarray<const uint32_t,4>,
ndarray<const uint64_t,1>, ndarray<const uint64_t,2>, ndarray<const uint64_t,3>, ndarray<const uint64_t,4>,
ndarray<const float,1>, ndarray<const float,2>, ndarray<const float,3>, ndarray<const float,4>, ndarray<const
double,1>, ndarray<const double,2>, ndarray<const double,3>, ndarray<const double,4>
std_string = include           # std::string
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# TYPE FILTER SHORTCUT
#
# In addition to filtering Psana types by the options above, one can use
# the type_filter option below. For example:
#
# type_filter = include cspad       # will only translate cspad types. Will not translate
#                                 # ndarrays or strings
# type_filter = exclude Andor evr   # translate all except the Andor or Evr types
#
# If you do not want to translate what is in the xtc file, use the psana shortcut:
#
# type_filter = exclude psana       # This will only translate ndarray's and strings
#
# Likewise doing:
#
# type_filter = include psana       # will translate all xtc data, but skip any ndarray's or strings
#
# The default is to include all
type_filter = include all
# note - if type_filter is anything other than 'include all' it takes precedence
# over the classes of type filter options above, like Cspad=include.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# SOURCE FILTERING
#
# The default for the src_filter option is "include all"
# If you want to include a subset of the sources, do
#
```

```
# src_filter include srcname1 srcname2
#
#  or if you want to exclude a subset of sources, do
#
# src_filter exclude srcname1 srcname2
#
# The syntax for specifying a srcname follows that of the Psana Source (discussed in
# the Psana Users Guide). The Psana Source recognizes DAQ alias names (if present
# in the xtc files), several styles for specifying a Pds Src, as well as detector matches
# where the detector number, or device number is not known.
#
# Unknown sources generate exceptions that by default stop the Translator. This can be
# inconvenient for users that reuse one configuration across many runs in an experiment,
# where some runs includes certain sources and some runs don't. You can tell the Translator
# to ignore unknown sources by setting the option
#
# unknown_src_ok=0   # to 1, by default it is False, which means stop.
#
# Specifically, format of the match string can be:
#
#       DetInfo(det.detId:dev.devId) - fully or partially specified DetInfo
#       det.detId:dev.devId - same as above
#       DetInfo(det-detId|dev.devId) - same as above
#       det-detId|dev.devId - same as above
#       BldInfo(type) - fully or partially specified BldInfo
#       type - same as above
#       ProcInfo(ipAddr) - fully or partially specified ProcInfo
#
# For example
#        DetInfo(AmoETOF.0.Acqiris.0)
#        DetInfo(AmoETOF.0.Acqiris)
#        DetInfo(AmoETOF:Acqiris)
#        AmoETOF:Acqiris
#        AmoETOF|Acqiris
#
# will all match the same data, AmoETOF.0.Acqiris.0. The later ones will match
# additional data (such as detector 1, 2, etc.) if it is present.
#
# A simple way to set up src filtering is to take a look at the sources in the
# xtc input using the psana EventKeys module.  For example
#
# psana -n 5 -m EventKeys exp=cxitut13:run=22
#
# Will print the EventKeys in the first 5 events.  If the output includes
#
#   EventKey(type=Psana::EvrData::DataV3, src=DetInfo(NoDetector.0:Evr.2))
#   EventKey(type=Psana::CsPad::DataV2, src=DetInfo(CxiDs1.0:Cspad.0))
#   EventKey(type=Psana::CsPad2x2::ElementV1, src=DetInfo(CxiSc2.0:Cspad2x2.1))
#   EventKey(type=Psana::Bld::BldDataEBeamV3, src=BldInfo(EBeam))
#   EventKey(type=Psana::Bld::BldDataFEEGasDetEnergy, src=BldInfo(FEEGasDetEnergy))
#   EventKey(type=Psana::Camera::FrameV1, src=BldInfo(CxiDg2_Pim))
#
# Then one can filter on these six srcname's:
#
#  NoDetector.0:Evr.2  CxiDs1.0:Cspad.0  CxiSc2.0:Cspad2x2.1  EBeam  FEEGasDetEnergy  CxiDg2_Pim
#
src_filter = include all
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# CALIBRATION FILTERING
#
# Psana calibration modules can produce calibrated versions of different
# data types. Depending on the module used, you may get an NDArray, an
# image, or the same data type as was in the xtc but with calibrated data.
#
# If you are doing the latter, the module output will be data of the same type
# and src as the uncalibrated data, with an additional key, such as 'calibrated'.
# If these modules are configured to use a different key, set calibration_key
# below accordingly:
calibration_key = calibrated
# The Translator defaults to writing calibrated data in place of uncalibrated
# data. If you do not want the calibrated data and would prefer to have the
```

```
# original uncalibrated data from the xtc, then set skip_calibrated to true.
skip_calibrated = false
# note, setting skip_calibrated to true will force sets exclude_calibstore
# (below) to be true as well.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# CALIBSTORE FILTERING
#
# Calibration modules may publish the data they used to produce the calibrated
# event objects. Examples of data would be pedestal values, pixel status (what
# pixels are hot) and common mode algorithm parameters. This data will be published
# in what is called the Psana calibStore. When the Translator sees calibrated
# event data, it will look for the corresponsinding calibStore data as well.
# If you do not want it to translate calibStore data, set the following to true.
exclude_calibstore = false
# otherwise, the Translator will create a group CalibStore that holds the
# calibstore data. Note, the Translator looks for all calibStore data associated
# with the calibration modules. If a calibration module was configured to not do
# certain calibrations (such as gain) but the module still put gain values
# in the config store (even though it did not use them) the Translator
# would still translate those gain values.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# KEY FILTERING
#
# Psana modules loaded before the translator may put a variety of objects in the event
# store. Be default, the Translator will translate any new data that it knows about.
# In addition to the psana types, it knows about NDArrays, C++ strings, and has a C++ interface
# for registering new simple types. NDarray's up to 4 dimensions of 10 basic types
# (8, 16, 32 and 64 bit signed and unsigned int, float and double) as well as the const
# versions of these types are translated.
#
# Generally Psana modules will attach keys to these objects (the keys are simply strings).
# To filter the set of keys that are translated, modify the parameter below:
key_filter = include all
# The default is to not look at the key but rather translate all data that the translator
# knows about. An example of including only data with the key finalanswer would be
#
# key_filter = include finalanswer
#
# To exclude a few keys, one can do
#
# key_filter = exclude arrayA arrayB
#
# Note, key filtering does not affect translation of data without keys. For instance
# setting key_filter = include keyA does not turn off translation of data without keys.
# Of all the data with keys, only those where the key is keyA will be translated.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# EVENTKEY FILTERING
#
# One can also filter data based on the event key. The default is
#
# eventkey_filter = include all
#
# but one could do
#
# eventkey_filter = exclude Frame__xtcav Frame__yag3
#
# that is, one can provide a list of event keys, which are (type, src, keystring) triplets, and
# the final key string is optional. The three items, type, src, keystring, must be separated by
# double underscores, i.e: __
# # # # # # # # # # # # # # # # # # # # #
# EXCLUDE CONFIGURE EVENT DATA
#
# this option was put in place to address a problem with a few runs in a particular
# experiment. Generally you do not need to turn this on. The explanation is as follows.
# Usually regular event data does not appear during the configure transition. The
# exception is Bld data. Each bld type will get an entry during the configure - more as
# a signal that the data is present - the values are not reliable. For this experiment,
# the BldDataSpectrometerV1 during the Configure transition had a junk number of peaks
# that caused psana to crash. To address this, one can set the following:
exclude_config_eventdata = false
```

```
# to true
# ---------------------------------------
# SPLIT INTO SEPARTE HDF5 FILES BASED ON CALIB CYCLES
#
# There are two reasons to split the Translator output into separate files based on
# calib cycles. The first is to reduce the size of the hdf5 files, and the second is
# to speedup translation by translating separate calib cycles in parallel. The default
# is to not split:
split = NoSplit
# however the Translator also supports SplitScan mode. This can only be invoked by running
# the separate driver program
#
#  h5-mpi-translate
#
# which requires MPI to be available in the environment. Under the hood, it will use the two other
# values for the split parameter - MPIWorker and MPIMaster - but users should not set these directly.
# In SplitScan mode, in addition to the output File, separate files will be made for the calib cycles.
# The output file (the master file) will include external links to the other files. Several mpi jobs are
# run simultaneously to divide the work of creating the calib cycle files. For example, running six jobs
# to produce out.h5 might look like:
#
#   mpirun -n 6 h5-mpi-translate -m Translator.H5Output -o Translator.H5Output.output_file=out.h5 exp=xppd9714:
run=16
#
# The driver program, h5-mpi-translate, takes all arguments that psana takes.
# If six jobs were used, one becomes the master process and the other five are the workers.
# The master process does two things. First it writes the file out.h5 with the external links
# to the calib files. Second it reads through all the data and finds the calib cycles. When it
# finds a calib cycle, it tells the next available worker where this is. When a worker is done,
# it tells the master process. The master process than adds all neccessary external links from
# out.h5 to the translated calib file produced by the worker.
#
# Generally, there will be one calib cycle file for each calib cycle. However to prevent to many
# calib cycle files from being produced for experiments that have only a few events per calib cycle,
# an option controls the minumum number of events per external calib cycle file. The default is
# min_events_per_calib_file = 100
# For example, if there are only 10 events per calib cycle, and assuming the master file is called
# out.h5, the file output_cc0000.h5 will contain the groups
#
# /CalibCycle:0000
# /CalibCycle:0001
# ...
# /CalibCycle:0009
#
# and the file output_cc0010.h5 will start with group /CalibCycle:0010
#
# As mentioned above, when workers finish a calib cycle file, they send a message to the master.
# How frequently the master stops reading through the data to check for these messages is controlled
# by the following parameter
# num_events_check_done_calib_file = 120
# that is, it defaults to check for a 'done' message from a worker every 120 events.
# by default, the calib cycle files are written to the same directory as the master file. Optionally,
# they can be placed into a subdirectory based on the master filename. The subdirectory name is the
# master file basename, without the extension, with _ccfiles appended to it. This subdirectory will be
# created if need be. To do this, set
#
# split_cc_in_subdir = True
#
# then if one does something like
#
# output_file = mydir/xpptut13-r0179.h5
# split_cc_in_subdir = True
#
# one will get
#
# mydir/xpptut13-r0179.h5
# mydir/xpptut13-r0179_ccfiles/xpptut13-r0179_cc0000.h5
# mydir/xpptut13-r0179_ccfiles/xpptut13-r0179_cc0001.h5
# ...
#
# rather than
```

```
# mydir/xpptut13-r0179.h5
# mydir/xpptut13-r0179_cc0000.h5
# mydir/xpptut13-r0179_cc0001.h5
##
# When running the h5-mpi-translate and specifying user psana modules (perhaps to add ndarrays
# into the translation or dynamically filter events) it is important to note that these modules
# are restarted for each calib cycle file. That is these modules will have their beginJob/endJob
# and beginRun/endRun routines called for each calib file that a worker produces.
#
#### FAST INDEX ####
#
# For online analysis with live data, one of the impediments to keeping up with the data is the time
# it takes h5-mpi-translate's to read through the data to find the calib cycles. As long as users
# read the small data (adding :smd to the dataset specification) h5-mpi-translate should have no
# trouble indexing the calib cycles in real time.
#
# If for some reason there is a problem with the small data, users can fall back on the
# fast_index feature against the large xtc files, however be aware that the web portal
# adds :smd to the datasource, so you will have to coordinate with data management to
# switch to the large xtc. Below we document fast_index.
#
# fast_index takes advantage of the unique signature of each
# new calib cycle, combined with the regular structure of the separate xtc data files in order to
# limit the reading to just one of the file. In this way, the h5-mpi-translate master rank
# need only get through the data it reads/searches at 20hz to keep up with the data. Part of why
# it is deprecated is because it is not guaranteed to work, whereas small data will.
#
# The translator supports the following options to turn on fast indexing and controlling how much
# time is spent searching the other files
#
# fast_index_force=0                 # set to 1 to turn fast indexing on
# fi_mb_half_block=12          # when fast indexing is on, use 12MB on each side, or 24MB for each block that
is searched
# fi_num_blocks=50             # this it half the number of 'other' blocks to try. The translator will try 1 +
2*50 = 101 blocks if this is 50
#
# Some details, If the Translator finds a calib cycle at offset N in DAQ stream 0, then the Translator
# will by default look in a 24MB block around offset N in stream 1, i.e., N -+ 12MB. It is looking for a
# match on about 52 bits, spread out among 20 bytes. If the Translator fails to find the calib cycle in
# those 24MB, then it tries the next 24MB below, then the next 24MB above, then below again, then above
# again, etc. In the end, the Translator will cover 5` of these blocks, or 51*24MB=1.2GB in stream 1.
# After it finds the calib cycle in stream 1, it repeats this process for stream 2,3,4 and 5.
# If the Translator fails with any of these streams, it throws an exception.
#
# Another option related to split scan is
#
# first_calib_cycle_number
#
# which is a 0-up counter for the first calib cycle that the MPIWorker will see. However users should not set
# this option - it is set by the Translator.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# COMPRESSION
#
# The following options control compression for most all datasets.
# Shuffling improves compression for certain datasets. Valid values for
# deflate (gzip compression level) are 0-9. Setting deflate = -1 turns off
# compression.
shuffle = true
deflate = 1
# if deflate is set to -1, set shuffle to false, as it performs no function without compression.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# TECHNICAL, ADVANCED CONFIGURATION
#
# ----------------------------------------
# CHUNKING
# The commented options below give the default chunking options.
# Objects per chunk are selected from the target chunk size (16 MB) and
# adjusted based on min/max objects per chunk, and the max bytes per chunk.
# It is important that the chunkCache (created on a per dataset basis) be
# large enough to hold at least one chunk, ideally all chunks we need to have
```

```
# open at one time when writing to the dataset (usually one, unless we repair
# split events):

# chunkSizeTargetInBytes = 1703936 (16MB)
# chunkSizeTargetObjects = 0 (0 means select objects per chunk from chunkSizeInBytes)
# maxChunkSizeInBytes = 10649600  (100MB)
# minObjectsPerChunk = 50
# maxObjectsPerChunk = 2048
# chunkCacheSizeTargetInChunks = 3
# maxChunkCacheSizeInBytes = 10649600  (100MB)
# By default, the Translator looks for control data to see if the number of events is known.
# If so, this overrides options above. To control chunking, one should also set useControlData
# below to 0 (or False)
# useControlData = 1
#
# ---------------------------------------
# REFINED DATASET CONTROL
#
# There are six classes of datasets for which individual options for shuffle,
# deflate, chunkSizeTargetInBytes and chunkSizeTargetObjects can be specified:
#
# regular (most everything, all psana types)
# epics (all the epics pv's)
# damage (accompanies all regular data from event store)
# ndarrays (new data from other modules)
# string's (new data from other modules)
# eventId (the time dataset that also accompanies all regular data, epics pvs, ndarrays and strings)
#
# The options for regular datasets have been discussed above. The other five datasets
# get their default values for shuffle, deflate, chunkSizeInBytes and chunkSizeInObjects
# from the regular dataset options except in the cases below:

# damageShuffle = false
# stringShuffle = false
# epicsPvShuffle = false
# stringDeflate = -1
# eventIdChunkSizeTargetInBytes = 16384
# epicsPvChunkSizeTargetInBytes = 16384
# The rest of the shuffle, deflate and chunk size options for the other five datasets are:
#
# eventIdShuffle = true
# eventIdDeflate = 1
# damageDeflate = 1
# epicsPvDeflate = 1
# ndarrayShuffle = true
# ndarrayDeflate = 1
# eventIdChunkSizeTargetObjects = 0
# damageChunkSizeTargetInBytes = 1703936
# damageChunkSizeTargetObjects = 0
# stringChunkSizeTargetInBytes = 1703936
# stringChunkSizeTargetObjects = 0
# ndarrayChunkSizeTargetInBytes = 1703936
# ndarrayChunkSizeTargetObjects = 0
# epicsPvChunkSizeTargetObjects = 0
# ---------------------------------------
# SPLIT EVENTS
# When the Translator encounters a split event, it checks a cache to see
# if it has already seen it.  If it has, it fills in any blanks that it can.
# To prevent this cache from growing to large, set the maximum number of
# split events to look back through here (default is 3000):
max_saved_split_events = 3000
```

Translation and Damage

psana has a specific damage policy that tells it what damaged data is acceptable for psana modules and what data is not. The default behavior is

- configStore - only undamaged data is stored in the configStore
- EventStore - undamaged data, and EBeam data with user damage is stored in the event, all other damage is not stored

the translator records event ids and damage for any xtc data that passes psana's damage policy. So by default, damaged config objects, and damaged events (other then user damaged EBeam data) are not translated. This deviates slightly from what o2o-translate would translate. o2o-translate would also store out of order damaged event data. There is a psana option that can be added to the [psana] section of the .cfg file to recover this behavior. Below we document some special options that control what damaged data psana stores:

- store-out-of-order-damage - defaults to false, set to true if you want to translate out of order damaged data
- store-user-ebeam-damage - defaults to true, set to false if you do not want to translate EBeam data that only has user damage
- store-damaged-config - defaults to false, set to true if you want to store damaged config data