Python Scripting

Environment Setup

Three C++ library extensions to Python have been written {"pycdb", "pydaq", "pyami"}

which allow scripting of interactions with the data acquisition and online monitoring system. The necessary environment changes to pickup these extensions are:

```
#!/bin/csh
setenv DAQREL /reg/g/pcds/dist/pds/7.0.2/build  (the latest release goes here)
setenv PYTHONPATH ${PYTHONPATH}:${DAQREL}/pdsapp/lib/x86_64-linux:${DAQREL}/ami/lib/x86_64-linux
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${DAQREL}/pdsdata/lib/x86_64-linux:${DAQREL}/pdsalg/lib/x86_64-
linux:${DAQREL}/pdsapp/lib/x86_64-linux:${DAQREL}/ami/lib/x86_64-linux:${DAQREL}/build/pds/lib/x86_64-
linux:${DAQREL}/build/qt/lib/x86_64-linux
```

Editing Configurations - the 'pycdb' module

An example script for editing DAQ configurations follows:

```
import pycdb

def upgradeIpimbConfig(cfg):
    dat = cfg.get()          # retrieve the contents in a Python dictionary object
    dat['trigPsDelay'] = 0   # reset the presampling delay
    cfg.set(dat)             # store the change

if __name__ == "__main__":
    db = pycdb.Db('/reg/g/pcds/dist/pds/xpp/configdb/current') # set the target database
    xtclist = db.get(alias="BEAM",typeid=0x20017)              # retrieve all IPIMB configurations for BEAM
runs (0x2 = version, 0x0017 = IPIMB)
    for x in xtclist:                                          # loop over the retrieved configurations
        upgradeIpimbConfig(x)                                  # modify the configuration
        db.set(x,"BEAM")                                       # write the configuration back to the database
    db.commit()                                                # update the database with the changes
```

The complete pycdb module programmer's description:

```
 --- pycdb module ---

class pycdb.Db(path)

        Initializes access to the configuration database residing at 'path'.

Members:

        get(key=<Integer> or alias=<string>,
            src=<Integer> or level=<Integer>, typeid=<Integer>) or
        get(key=<Integer> or alias=<string>,
            typeid=<Integer>)
                Returns a tuple of configuration datum which satisfies the
                search criteria.  The search criteria is composed of either
                a particular database key number (integer) or alias name
                (for example, "BEAM"), an integer detector id 'src' or level
                (for example, 0=control), and an integer typeid
                (for example, 0x00050008 = Evr configuration version 5).
                The Python type of each returned object is specific to the
                datum returned { pycdb.DiodeFexConfig, pycdb.CspadConfig, ... }.

        set(datum, alias)
                Inserts the configuration 'datum' into the database for the
                global entry 'alias' (for example, "BEAM").

        commit()
                Updates all current run keys with the data inserted via calls to 'set'.

        clone(key)
                Returns a new database key created as a copy of the existing key 'key'.
                The returned key is suitable for modifying via calls to 'substitute'.

        substitute(key, datum)
                Insert the configuration 'datum' only for the given database 'key'.
```

Controlling the DAQ - the 'pydaq' module

An example script for controlling the DAQ through a scan follows:

```
import pydaq

if __name__ == "__main__":
    host = 'xpp-daq'     # host name running the DAQ control process
    platform = 0         # DAQ identifier
    cycles = 100         # number of iterations in the scan
    nevents = 105        # number of events at each iteration
    do_record = False    # option to record the data
    daq = pydaq.Control(host,platform)                         # Connect to the DAQ control process

    daq.configure(record=do_record,                            # Configure the DAQ with the scan information,
                  events=nevents,                              # number of events per scan cycle,
                  controls=[('EXAMPLEPV1',0),('EXAMPLEPV2',0)]) # list of scan variables and current values

    print "Configured."

#
#  Wait for the user to declare 'ready'
#    Setting up monitoring displays for example
#
    ready = raw_input('--Hit Enter when Ready-->')

    for cycle in range(options.cycles):                        # Loop over the scan cycles
        print "Cycle ", cycle
        pv1 = cycle                                            # Change the scan variable
        pv2 = 100-cycle                                        #   For example, move a motor with EPICS.
        daq.begin(controls=[('EXAMPLEPV1',pv1),                # Acquire the events with the
                            ('EXAMPLEPV2',pv2)])               #   list of scan variables and their values.
        # enable the EVR sequence, if necessary
        daq.end()                                              # Wait for the events to complete
        # disable the EVR sequence, if necessary

    if (do_record==True):
        print 'Recorded expt %d run %d' % (daq.experiment(),daq.runnumber())

#
#  Wait for the user to declare 'done'
#    Saving monitoring displays for example
#
    ready = raw_input('--Hit Enter when Done-->')
```

The previous script simply reports the change in the scan variables to the DAQ and acquires the indicated number of events for each scan point. A more complete implementation would include code that controls the motors/devices that the scan variables represent (through 'pyca', for example). Alternatively, the scan may be designed to vary the DAQ configuration (a trigger delay scan, for example); this can be accomplished in conjunction with the 'pycdb' module. In addition, the implementation may wish to retrieve the accumulated data at each step for scripted processing or plotting; this is described in the next section ('pyami' module).

The complete pydaq module programmer's description:

```
 --- pydaq module ---

class pydaq.Control(host, platform=0)

    Arguments:
        'host'     : host name or IP address (dotted-string notation or integer)
        'platform' : DAQ platform number (subdivision of readout hardware)

    Function:
        Initializes the interface for controlling the data acquisition system remotely.
        Creates a connection with the local control and queries the configuration
        database and current key in use.

Members:

    Control.dbpath()
        Returns path to configuration database

    Control.dbkey()
        Returns current configuration key (integer) in use
    Control.dbalias()
        Returns current configuration alias (string) in use

    Control.partition()
        Returns a list of dictionary objects describing all nodes in the DAQ readout.

    Control.configure(record=<Bool>,
              key=<Integer>,
              events=<Integer> or l1t_events=<Integer> or l3t_events=<Integer> or duration=[seconds,
nanoseconds],
              controls=[(name,value)],
              monitors=[(name,lo_range,hi_range)],
              partition=[()])
        Configures control to use database key (default to current key) either:
          (1) collecting a fixed number of events on each cycle (when events=<Integer>
              or l1t_events=<Integer> is specified) or
          (2) collecting events until a fixed number of events have been accepted by
              the level3 filter (when l3t_events=<Integer>) or
          (2) collecting events for a fixed duration (when duration=[seconds,nanoseconds]
              is specified).
        The list of variables controlled (optional) in the scan and
        the list of variables to be monitored (optional) during acquisition
        are specified.
        The option to record can also be set.  If it is omitted, then the value from
        the local control is used.
        A modified list of objects from the Control.partition() call may be given for the partition argument to
        select only a subset of detectors for readout or recording by changing the values of the 'Readout' or
'Record'
                dictionary entries.  If this argument is omitted, the partition is readout and recorded as
initially configured.

    Control.begin(events=<Integer> or l1t_events=<Integer> or l3t_events=<Integer> or duration=[seconds,
nanoseconds],
              controls=[(name,value)],)
              monitors=[(name,lo_range,hi_range)])
        Begins acquisition for the specified settings of control variables (if specified).
        Actual control of these variables is the responsibility of the remote application.
        Monitor variables (optional) are enforced during acquisition.  Omitted values
        default to those specified most recently (or in the 'configure' method).

    Control.end()
        Waits for the end of acquisition cycle signalled from the local host control.

    Control.stop()
        Signals the local host control to terminate the current acquisition cycle.
                This method can be used to prematurely end a scan without closing the connection or
reconfiguring.
                The KeyboardInterrupt(SIGINT) signal handler can be reimplemented to call this method, which
will
                result in the scan ending and a python ValueError exception being raised.

    Control.eventnum()
        Returns the number of events acquired in the current acquisition run.

    Control.experiment()
        Returns experiment number of run, if recorded.
        Only valid after acquisition is complete.

    Control.runnumber()
        Returns run number, if recorded.
        Only valid after acquisition is complete.
```

Monitoring the Data - the 'pyami' module

An example script (not very useful) for retrieving data accumulated by the monitoring follows:

```
import pyami

class AmiScalar(pyami.Entry):      # subclass for readability
    def __init__(self,name):
        pyami.Entry.__init__(self,name)

class AmiAcqiris(pyami.Entry):     # subclass for readability
    def __init__(self,detid,channel):
        pyami.Entry.__init__(self,detid,channel)

eth_lo = 0x7f000001
eth_mc = 0xefff2604
CxiAcq = 0x18000200                # detector identifier for CxiEndstation Acqiris readout

if __name__ == "__main__":
    pyami.connect(eth_mc,eth_lo,eth_lo)     # example parameters for a monitoring playback job

    x = AmiScalar("ProcTime")               # accumulate (events,mean,rms) for 'ProcTime' scalar variable
    x.get()                                 # return accumulated data

    x = AmiAcqiris(CxiAcq,1)                # accumulate averaged waveform for Cxi Acqiris readout module
    x.get()                                 # return accumulated waveform
```

A "complete" scripted example for scanning, acquiring data, and plotting the data is scan_plot.py .

The pyami module programmer's description:

```
 -- pyami module --

connect(Server_Group, CDS_interface, FEZ_interface)
    Connects the module to the group of monitoring servers.
    The input parameters are specific to the hutch.
    If the last two parameters are omitted, they will be learned from the
    available network interfaces on the host.
    This must be called before any of the following class methods.

connect(PROXY_ip_address, CDS_interface)
    Connects the module to the group of monitoring servers known by the PROXY.

discovery()
    Returns a list of detectors available for monitoring.

list_env()
        Returns a list of scalar variables available for monitoring.

set_l3t(filter_string)
    Sets the current DAQ L3T filter to <filter_string>.  See "event filter" below.

clear_l3t()
    Removes the DAQ L3T filter.

class pyami.Entry(name) or
class pyami.Entry(name,'Scalar')
    Monitors data from the scalar variable 'name'.  A dictionary of
    (type='Scalar', time=<last event time in seconds since the Epoch>,
     entries=<number of events accumulated>, mean=<value>, rms=<value>) is accumulated.

class pyami.Entry(name,'TH1F',nbins,xlo,xhi)
    Monitors data from the scalar variable 'name'.  A dictionary of
    (type='TH1F', time=<last event time in seconds since the Epoch>,
     uflow=<number of underflows>, oflow=<number of overflows>, data=( n_bin0, n_bin1, ...)) is accumulated.

class pyami.Entry(name,'Scan',xvariable,nbins)
    Monitors data from the scalar variable 'name'.  A dictionary of
    (type='Scan', time=<last event time in seconds since the Epoch>,
        nbins=<value>, current=<index of most recent entry>, xbins=( x0, x1, ...), yentries=( n0, n1, ...),
        ysum=( y0, y1, ...), y2sum=( y2_0, y2_1, ...)) is accumulated; where
    'current' is the bin with the most recent entry, 'xbins' is a list of unique x-variable values,
    'yentries' is a list of the number of summed entries with that x-variable value,
    'sum' is a list of the sum of y-variables values in each bin, 'y2sum' is a list of the sum
    of y-variable values squared in each bin.  The lists are of length 'nbins' with only the most
    recent entries retained.

class pyami.Entry(det_identifier) or
class pyami.Entry(det_identifier,channel)
    Monitors the data from the detector associated with 'det_identifier' and
    'channel'.  A dictionary of data averaged over events is accumulated.  The
    dimensions and binning of the data are determined from the detector.
    The dictionary format is
    (type='Waveform', time=<last event time in seconds since the Epoch>
        entries=events, xlow, xhigh, data=(y0, y1, ...)) or
    (type='Image', time=<last event time in seconds since the Epoch>,
        entries=events, offset=dark_level, ppxbin, ppybin,
        data=((row0col0,row0col1,...),(row1col0,row1col1,...))) or
    (type='ImageArray', time=<last event time in seconds since the Epoch>,
        entries=events, offset=dark_level, ppxbin, ppybin,
        data=( ((row0col0,row0col1,...),(row1col0,row1col1,...))_0,
               ((row0col0,row0col1,...),(row1col0,row1col1,...))_1,
               ... ))

    "event filter"
    Each of the above methods also takes an optional final string argument that
    defines an event filter.  The string must take the form:

        _lo_value_<_scalar_name_<_hi_value_  or
        (_expr1_)&(_expr2_)                  or
        (_expr1_)|(_expr2_)

    where _expr1_ and _expr2_ also take one of the forms.

Members:

    Entry.get()
        Returns the data accumulated since the object was created.
    Entry.clear()
        Resets the data accumulation.
    Entry.pstart()
        Start continuous mode accumulation ("push" mode).
    Entry.pstop()
        Stop continuous mode accumulation.
    Entry.pget()
        Return continuous mode data.

"Push" mode description:

    The event data is "pushed" to the python module's application by the monitoring servers once the Entry.
pstart() command is given.  The Entry.pget() call simply waits for the data to arrive, if necessary, and
returns it to the caller.  The Entry.pstop() command instructs the servers to cease sending the data.  Note
that this mode can easily consume considerable network bandwidth on the local machine.


class pyami.EntryList([list of arguments that satisfy pyami.Entry above])
     Defines a list of objects to access using the same resources as one pyami.Entry object.  The list of
objects is accessed synchronously, so each object's accumulation of events will be the same.

Members:

    EntryList.get()
        Returns a list of the data accumulated since the object was created.
    EntryList.clear()
        Resets the data accumulation.
    EntryList.pstart()
        Start continuous mode accumulation ("push" mode).
    EntryList.pstop()
        Stop continuous mode accumulation.
    EntryList.pget()
        Return a list of the continuous mode data.
```