EPICS Vv4 CORE ACCELERATOR SERVICES

Greg White for EPICS v4 team, v1.0 30-Jul-2010, following meeting at BNL.

This version 1.04, 29-Sept-2010.

Table of Contents

Introduction

This is an outline proposal for the minimum set of particle accelerator control software services required to support high level physics software. It's intended to be a discussion document to focus discussion on the standard interfaces of these services.

Accelerator services are envisioned to be part of an upcoming EPICS "Vv4" effort. The purpose of this effort is to agree on a minimum set of those services and their APIs, so that implementations of both the data services and the physics software using them, can proceed independently.

Importantly, the standard interface of a service is intended to be a minimum. Service implementations may well add more features and returned data.

Examples of the use cases of physics software using these services would be such things as BPM orbit display, orbit correction ("Steering"), Linac Energy Management (LEM), bumps, dispersion correction, and parts of the feedback problem.

Some of the interfaces in this initial proposal are based on experience with those presently in the Accelerator Independent Data Access system (AIDA) [1], where some clarification may be sought.

The Minimum Service Set

Minimum set of service interfaces for accelerator applications, described in this document;

1. Accelerator Model
2. BPM Orbit Data
3. Magnet readback/control
4. Klystron readback/control
5. Archive History

In general, that minimum set would cover a large proportion the data and control requirements of physics applications. Others also very useful;

6. General Relational Database (RDB)
7. Linac Energy Management
8. Control Point Configuration ("Snapshot") Management.

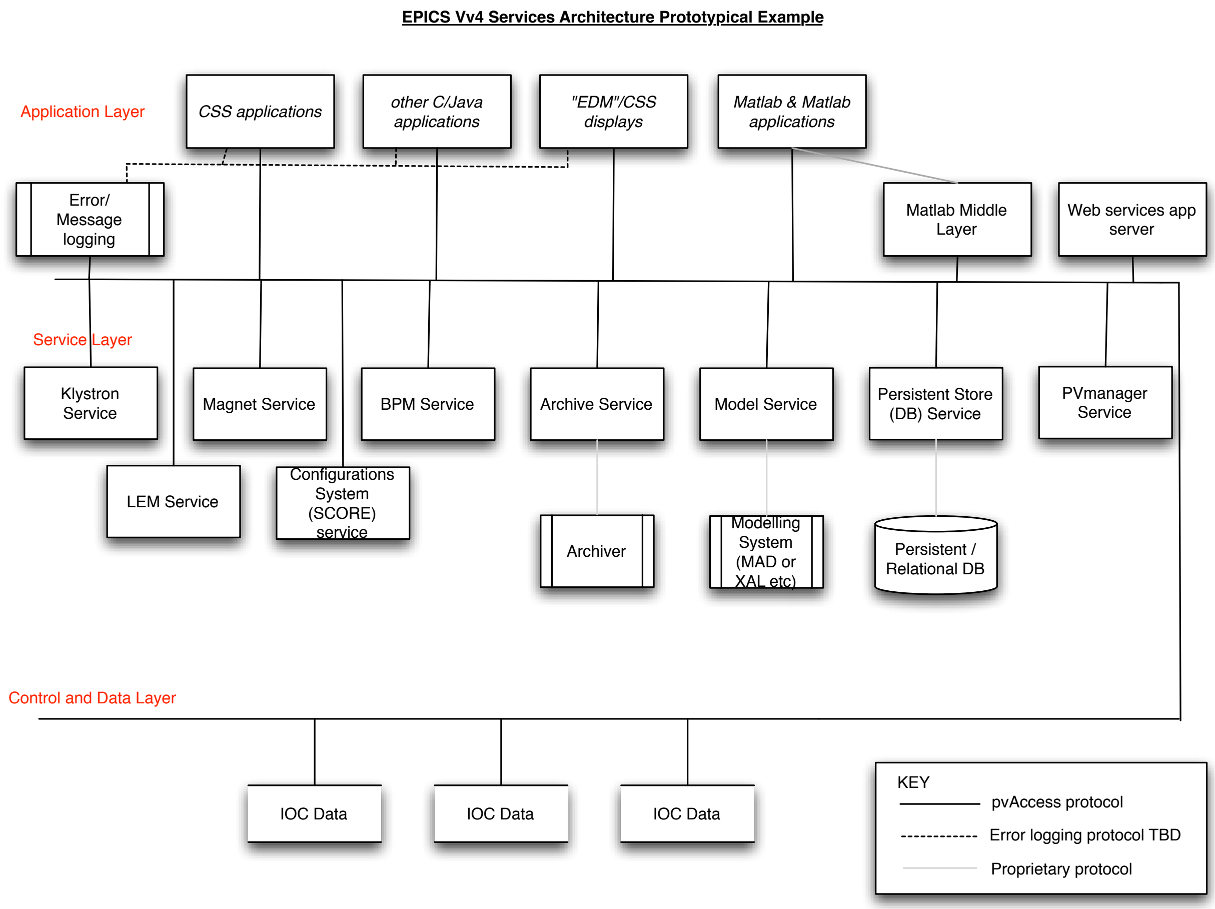

Figure 1: Where accelerator data services fall in the basic architecture, showing that it's the basic 3 tier model.PVmanager is likely to be used by many of the application oriented services.

Properties of Accelerator Services and their clients

There are some important properties of accelerator services, the structure of the data they provide, and the clients which consume them, that make them significantly different from the control data normally associated with EPICS:

1. Services deal with beamlines and systems of devices at a time. For instance, rather than getting the values for a single device at a time, the BPM service gets the orbit offsets for all the monitors in a beamline. The klystron service deals with all the klystron in a machine wide klystron complement. Same for the magnet service, etc.

2. Service data I/O is typically of “structured” or “semi-structured”, or “complex”. Whatever you call it, it’s most often not a single value or an array, it’s typically a heterogenous structure of arrays of values or heterogenous structures.

3. Synchronous is common. The reason for this significant difference to the more common asynchronous control API, are:
    I. High level services are parameterized (they take user input data)
    II. There is often significant computation or backend lookup required
    III. The client really can't do anything other than service GUI requests until it has the results. So the GUI may well ask for the data from the service in a non-GUI thread, but the service itself is synchronous.

4. Services use other services. For instance the BPM service will use the configurations service to get and set the reference orbit; the LEM service would use the model, magnet and klystron services.

5. The central service is the Name Service. Since almost all such services deal not in individual devices, but beamlines or named sets of devices, services must be able to expand named sets into their composite devices very efficiently. So the primary service is the Name Service.

6. The names known to the Name Service are typically from the accelerator model! Since the only really reliable list of which devices are in a beamline, the names in the name service are derived heavily from the model. When a model defines new element, the name service has to be updated immediately.

7. Errors have to be dealt with systemically, so that effects can be traced to causes over the different services. Error messages emanating from all parts of the system (from IOC to matlab scripts) must be delivered to a single logging system, and those errors must be in constant view of the users of the physics applications themselves.

Model Service

Provides the basic linear accelerator model data for primary diagnostics and optimization programs.

Interface Summary

Gets:

1. Twiss model data for given beamline element
2. R-matrix model data
   a. for a given beamline element (from start of beamline[1])
   b. between given beamline elements A and B (so called “Rmat A to B”)
3. Tables of whole beamlines’ Twiss and R matrix data.

Sets:

Not necessary[2].

This analysis does not cover the emerging use case of “start to end simulation” in which it makes more sense for model data to be continuously computed. But the requirement of online optimization programs for such systems is far less than their need for basic release controlled, linear envelope model data (Twiss and R-mats).

The following subsections define the 3 modes of model data acquisition in detail.

Twiss parameters for given lattice element

Returns the 11 Twiss parameters (aka Courant-Snyder) params, plus the S position and effective length, slice effective length, and ordinal position in the beamline, of the element in question. Include S and lengths since only the model is the authoritative source and they can vary by model even of the same beamline.

Use cases

[1] In practice the start of a beamline may or may not be the start of the accelerator. It may be the start of the accelerator section in which the device is situated, or dependent on other parameters. The precise interpretation of the returned data will vary from institution to institution and should not be defined here.

[2] Set via EPICS pvaccess need not be supported for any but the most sophisticated online model systems. This is for a few reasons: first, model computation and upload will be many, many times less frequent than download. Secondly, it’s likely that the model data output of a modelling code would be in some persistent form like files or a relational DB, so upload via pvaccess to a server is largely redundant because the server would go straight to the source.

Control room lattice diagnostics, emittance calculation, phase space “B-mag” diagnostics.

Options

```
[ RUN = <run-identifier> ]
      run-identifier := GOLD | LATEST | unspecified
```

Each of the 3 output data types for modelling (above), are parameterized by an optional identifier which specifies the modelling program[3] run that produced the model. The identifier most used is the "GOLD.” This GOLD (or sometimes called "golden") model, is the modelling run identified by control room operations, as that most descriptive of the present operating conditions, or which describes the objective operational conditions. Different labs may have different names for this "best" short term model. LATEST is the last model calculated for the TYPE (below). The RUN may also define an exact modelling run, like “4254” - which would indicate that the output of the 4254th modelling run is desired.

```
[ POS={BEG|MID|END} ]
```

Position within a thick device for which Twiss data is required. Permitted values are BEG (beginning), MID (middle), END (end). Default = END. For thin devices (monitors, correctors etc) this option is ignored.

```
[ TYPE={EXTANT|DESIGN} ]
```

Whether the model of the extant machine is desired, or the design model. The precise interpretation of “extant” will be vary by institution - but basically it refers to a model calculated from existing lattice element readback settings.

Returned Data Type Definition for Twiss parameters:

twiss double [15]: Kinetic Energy (GeV), psi x, beta x (m), alpha x (rad), eta x (m), eta x' (rad), psi y, beta y (m), alpha y (rad), eta y (m), eta y' (rad), S (m), effective length (m), slice effective length [0.0 for thin element], ordinal position in the beamline.

Examples for Twiss parameter acquisition for a single lattice element:

[3] A modelling program is often referred to a modelling “code”. Examples are MAD, Transport, XAL probe.

Example 1: Get twiss for some quad (a quad is thick, so the default will be to get the twiss at the end), from the extant “gold” model (the defaults).

```
> pvget QUAD:LI21:271/twiss

       0.25

   13.272953

    2.543171

   -0.385537

   -0.005005

   -0.006495

   15.122584

   31.135721

   23.249401

        0.0

        0.0

  2053.03235

      0.108

      0.054

      467.0
```

Example 2: Get twiss for the beginning of same quad, from the golden design model.

```
> pvget QUAD:LI21:271/twiss -o POS=BEG -o TYPE=DESIGN

       0.25

   13.231149

    2.710526

    1.984116

   -0.004533

   -0.002313

   15.119276

   33.161709

   -5.091002

        0.0

        0.0

  2052.92435

      0.108

        0.0

      463.0
```

The transfer matrix (6x6) for a given lattice element

Returns the 6x6 transfer matrix of the given named lattice element (plane coupled if necessary), either from the start of the beamline (1), or from another given lattice element (2):

(4) R matrix (6x6) from from start of beamline

If no "B" parameter is given, then the transfer matrix returned should be the R matrix from the the beamline start to the given lattice element (or in case of circular machine, the C matrix from injection to given element)

(5) R matrix (6x6) from the given lattice element, to another

If a “B” parameter is given, the transfer matrix returned should be the R matrix from the given “A” to the device “B”. This implements “R-mat A to B”, which is a ubiquitous requirement of online accelerator optimization software.

If the beamline is a transport line, then the 36 Rmat 6x6 should be interpreted as a linear transport matrix (i.e. an R aka T matrix). If the beamline is a ring, then the 6x6 should be interpreted as a closed orbit transfer matrix (i.e. a C matrix)

Use cases

This is the basic interface for building orbit steering software, bump, feedbacks etc. Eg this is the way you get the "R12" of a device.

Input Options

[RUN = <run-identifier>]

as defined above for Twiss parameters.

[POS = {BEG|MID|END}]

as defined above for Twiss parameters

[TYPE = {EXTANT|DESIGN}]

as defined above for Twiss parameters

[B = lattice-device-name]

String name of device name. Rmatrix returned will be from the query device (A) to this B.

Returned Data Type Definition:

Rmat: double[6][6]

R11, R12, R13, R14, R15, R16

R21, R22, R23, R24, R25, R26

R31, R32, R33, R34, R35, R36

R41, R42, R43, R44, R45, R46

R51, R52, R53, R54, R55, R56

R61, R62, R63, R64, R65, R66

Examples for R-matrix model data acquisition for a single element:

Example 1: Get the 6x6 transfer matrix for a beam position monitor, from the extant machine gold model.

Rmat: double[6][6]

Example 2: Get the R matrix for the middle of the quad in latest design model

```
> pvget QUAD:LI21:271/R -o TYPE=DESIGN -o POS=MID -o RUN=LATEST
    0.23      0.1234   0.0       0.0       0.067562 0.001167
   -0.34520   0.0923   0.0       0.0       0.046981 0.001514
    0.0       0.0     1.881007  4.857304 0.0       0.0
    0.0       0.0    -1.50064  -3.862346 0.0       0.0
   -0.00132 -0.001129 0.0       0.0       0.224701 0.003894
    0.162595 0.10285   0.0       0.0     -19.603   -0.233109
```

Example 2: Orbit Correction (Steering).

Acquire BPM orbit data and transfer matrix model of all correctors to all bpms:

```
corrNames[] = <get list of all corrs from pvmanager or by whole model acq below.>
bpmNames[] = <get list of all bpms from pvmanager or by whole model acq below.>
Ncor = corrNames.length
b = pvget("TAXXALL/BPMS).get(2);  // Get X orbit
Mbpm = bpmNames.length
for bpmi = 1, Mbpm
   for corj = 1, Ncor
       A[bpmi, corj] = pvget(corrNames[corj]+"/R","B="+bpmNames[bpmi]);
   end
end
x = inv(A)*b
x = -x
pvput("MAGNETSET/BDES",x) // Implement solution in corrector magnets.
```

The 1st order model (Twiss and R-mats) of a whole beamline

Return both the 36 matrix elements of the 6x6 transfer matrix matrix, and Twiss parameters, of each element in a whole beamline. Ie a big table which describes the 1st order model of the whole beamline.

Use cases: Basis for the system matrix of orbit correction software. Also can be used for client side caching the whole model, making repeated model operations faster than using the element by element interface.

As for individual device R-matrices, if the beamline is a transport line, then the 36 Rmat 6x6 should be interpreted as a linear transport matrix (i.e. an R aka T matrix). If the beamline is a ring, then the 6x6 should be interpreted as a closed orbit transfer matrix (i.e. a C matrix).

You would have 1 such table for each modelling run of each beamline.

Input Options:

[RUN = <run-identifier>]

as defined above for Twiss parameters

[TYPE = {EXTANT|DESIGN}]

as defined above for Twiss parameters

Returned Data Type Definition:

Output a table, 1 row = 1 lattice element slice, arranged in phase advance beamline order from the beginning of the beamline. Table has the following columns:

1. long: Ordinal position of the lattice slice described by this row (as defined by phase advance)
2. string: EPICS PV device name, if there is one, of the lattice slice described by this row
3. string: The MAD or otherwise offline modelling code name, if different to 2 above
4. double: S position of the lattice slice described by this row
5. string: BEGIN|END. Where in the lattice element that this row pertains.

For thin elements this will simply be END. For thick elements, e.g. quadrupoles, will require >1 row, since the transfer matrix will be required at least at their BEGINning, after the first lattice slice, and at the end of the element. A typical quadrupole with an embedded beam position monitor that takes 2 lattice slices, would be represented by 4 output rows (see below). Klystrons with many cavities take a number of lattice slices, and concomitantly more rows. However, for online modelling we can probably simplify to the following convention. This handles 2 slice quads completely, and carries enough information about multi-cavity klystrons for online accelerator optimization purposes:

- return 2 table rows for the first slice of any thick element (for its beginning and end) and then 1 record for each slice thereafter. So, for instance a typical 2-slice quad with an embedded BPM, starting at beamline element position 12, would look something like this:

```
 12 QUAD:AREA1:234 QF1 12.232 BEG [data for entry to 1st q slice]
 13 QUAD:AREA1:234 QF1 12.321 END [data for exit of 1st q slice]
 14 BPM:AREA1:234  B4  12.321 END [data for the embedded monitor]
 15 QUAD:AREA1:234 QF1 13.43  END [data for the exit of 2nd slice]
```

6. columns 6-42, double[36]; the 36 elements of the 6x6 transfer matrix (interpreted as R or C depending on whether linac or circular) arranged so first 6 are the first row

43. columns 43-54 double[11]; the 11 Twiss parameters Kinetic Energy (GeV), psi x, beta x (m), alpha x (rad), eta x (m), eta x' (rad), psi y, beta y (m), alpha y (rad), eta y (m), eta y' (rad)

Examples:

Example 1: Get the twiss and rmats of all of the lattice elements in the beamline group named “fullmachine”, in model run 2345.

For illustration, the following is an extract of only the first 41 columns of the element model data, taken from beamline device 92 onwards for the next 8 rows (lattice elements). The full table would be 54 columns and possibly 100s of rows.

```
> pvget fullmachine/twissandrmats type=extant -o run=2345
```

```
   98.0  QA02        QUAD:IN20:371   BEGIN  2019.831226   -0.38839   0.645558  0.0  0.0       0.0       0.0  -0.542217   0.661894  0.0  0.0       0.0

0.0       0.0       0.0    -0.19205    0.264731  0.0       0.0       0.0       0.0  0.175233  -0.725584       0.0       0.0       0.0       0.0      0.0

0.0       1.0  0.012882       0.0       0.0  0.0  0.0       0.0   0.092959

   100.0  QA02       QUAD:IN20:371  MIDDLE  2019.885226  -0.411207   0.670628  0.0  0.0       0.0       0.0  -0.300573   0.264135  0.0  0.0       0.0

0.0       0.0       0.0    -0.18567    0.229657  0.0       0.0       0.0       0.0  0.061701  -0.576985       0.0       0.0       0.0       0.0      0.0

0.0       1.0  0.012883       0.0       0.0  0.0  0.0       0.0   0.092959

   101.0  BPM5       BPMS:IN20:371   BEGIN  2019.885226  -0.411207   0.670628  0.0  0.0       0.0       0.0  -0.300573   0.264135  0.0  0.0       0.0

0.0       0.0       0.0    -0.18567    0.229657   0.0       0.0       0.0       0.0   0.061701  -0.576985       0.0       0.0       0.0       0.0      0.0

0.0       1.0  0.012883       0.0       0.0  0.0  0.0       0.0   0.092959

   103.0   QA02       QUAD:IN20:371     END  2019.939226  -0.420676    0.67393  0.0  0.0       0.0       0.0  -0.049173  -0.142199  0.0  0.0       0.0

0.0       0.0       0.0    -0.18535    0.202078   0.0       0.0       0.0       0.0  -0.049817  -0.447219       0.0       0.0       0.0       0.0      0.0

0.0       1.0  0.012883       0.0       0.0  0.0  0.0       0.0   0.092959

   106.0  FLNGB1                     BEGIN  2020.005114  -0.423916    0.66456  0.0  0.0       0.0       0.0  -0.049173  -0.142199  0.0  0.0       0.0

0.0       0.0       0.0   -0.188632    0.172612   0.0       0.0       0.0       0.0  -0.049817  -0.447219       0.0       0.0       0.0       0.0      0.0

0.0       1.0  0.012883       0.0       0.0  0.0  0.0       0.0   0.092959

   107.0  L0B___1       ACCL:IN20:400   BEGIN  2020.005114  -0.423916    0.66456  0.0  0.0       0.0       0.0  -0.049173  -0.142199  0.0  0.0       0.0

0.0       0.0       0.0   -0.188632    0.172612   0.0       0.0       0.0       0.0  -0.049817  -0.447219       0.0       0.0       0.0       0.0      0.0

0.0       1.0  0.012883       0.0       0.0  0.0  0.0       0.0   0.092959

   108.0  L0B___1       ACCL:IN20:400     END   2020.06376  -0.422431   0.649504  0.0  0.0       0.0       0.0   -0.04792  -0.141919  0.0  0.0       0.0

0.0       0.0       0.0   -0.189593    0.144885    0.0       0.0       0.0       0.0  -0.048974  -0.442946       0.0       0.0       0.0       0.0

0.0       0.0       1.0  0.012884       0.0       0.0  0.0  0.0   -0.052956   0.090393

   109.0  L0B___2       ACCL:IN20:400   BEGIN   2020.06376  -0.422431   0.649504  0.0  0.0       0.0       0.0   -0.04792  -0.141919  0.0  0.0       0.0

0.0       0.0       0.0   -0.189593    0.144885         0.0       0.0       0.0       0.0  -0.048974  -0.442946       0.0       0.0       0.0       0.0

0.0       0.0       1.0  0.012884       0.0       0.0  0.0  0.0   -0.052956   0.090393

   110.0   DLFDB                     BEGIN   2020.06376  -0.422431   0.649504  0.0  0.0       0.0       0.0   -0.04792  -0.141919  0.0  0.0       0.0

0.0       0.0       0.0   -0.189593    0.144885     0.0       0.0       0.0       0.0  -0.048974  -0.442946       0.0       0.0       0.0       0.0

0.0       0.0       1.0  0.012884       0.0       0.0  0.0  0.0   -0.052956   0.090393
```

Beam Position Monitor (BPM) Orbit Service

Provides the BPM monitor readings (x and y offsets) for a whole beamline. The data returned may well be beam synchronous, though this description is independent of whether it is.

Use cases

BPM orbit plots, orbit fitting, orbit correction.

Interface Summary

Get:

At minimum the BPM service should return a table of the X and Y offsets from the calibrated center of each beam monitor (BPMs and Toroids) in a given named beamline (or beamline section). Each value should be accompanied by an RMS, giving the measured error (valued 0 if only 1 one-beam reading was made, for each BPM, to gather these data).

Options:

[MEASDEF = <meassurement-definition>]

A freeform identifier which may be used to identify a timing configuration. The semantics will vary by institution and implementation. Eg “MEASDEF = 1hz” meaning just return the last bpm data that was accumulated in a synchronous 1 hz buffer. Or “MEASDEF=Fredsmeasdef” meaning use the timing definition defined for Fred’s experiment. Each institution will define their own default interpretation.

[TYPE = {ABS | DIFF}]

Whether the returned data is the absolute offset from the center of the calibrated center of the beampipe, or the difference to the GOLDen orbit, or the difference to the configuration given in the CONF parameter. Default value is ABS.

[CONF=<identifier>]

The identifier of a bpm device configuration snapshot. The returned data will be the difference: returned data = extant data minus config data. Default should identify the GOLDen orbit, so TYPE = DIFF, with no CONF options given, means return the difference to the Golden orbit.

[N = +veinteger]

If N is given, the returned data should be the result of averaging over the number of acquisition cycles given. Default is 1.

Set:

No requirement for Set.

Note, this interface is independent of the question of whether all BPM readings were taken on the same pulse, or whether the RMS is over consecutive pulses/turns of ring etc. This same output interface can be used in all cases.

Returned Data Type Definition:

1. string: EPICS device name, if there is one, of the lattice slice described by this row
2. double: S position (m) of the lattice slice described by this row
3. double: X offset (mm) (null if Y only bpm)
4. double: Y offset (mm) (null if x only bpm)
5. double: TMIT (coulombs)
6. double: X rms (null if Y only bpm)
7. double: Y rms (null if x only bpm)
8. long: BPM status (codified cause of most important severity or alarm).

Example. The following example asks for all the bpm orbit data conforming to a timing measurement definition numbered 41 (only meaningful in the context of the FACET SLAC machine), averaging over 5 pulses. Admittedly a very boring example because we weren't running and I couldn't be bothered to make up fake data.

```
> pvget TAXXALL/BPMS -o MEASDEF=41 -o N=5
   BPMS:TA01:324  27.8916 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:325  27.8916 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:344  28.3364 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:345  28.3364 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:408  29.7864 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:409  29.7864 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:434  30.5428 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:435  30.5428 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:480  31.8887 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:481  31.8887 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:530  32.9011 0.0  0.0   0.0 0.0  0.0      0
   BPMS:TA01:531  32.9011 0.0  0.0   0.0 0.0  0.0      0
...
```

Magnet Service

This service allows for getting and setting the setpoints of a set of beamline magnets in a synchronized and controlled way.

Note: I think the magnet service is a specialization of pvmanager. The same may be true of the klystron service below.

Use Cases:

> Operational diagnostics, Orbit Correction (Steering), Linac Energy Management (LEM), slow and global feedbacks.

Interface Summary

Get.

Get the extant B-field property value of the magnets in a given list of magnet names or named beamline. The property may be the desired or readback property.

By named beamline:

```
<beamlineidentifier>/<device-type>:<attribute>
eg lcls/ycor:bdes
```

By list of names:

```
MAGNETGET/<attribute>
eg MAGNETGET/BDES
```

Set.

Set the desired B-field property value of a set of magnets, to the values in a given array. The magnets may be identified by a list of names or a named beamline.

By named beamline:

```
<beamlineidentifier>/<device-type>:<attribute>
eg lcls/ycor:bdes
```

By list of names:

```
MAGNETSET/<attribute>
eg MAGNETSET/BDES
```

Options:

[CONDITION = { ALL | SOME }]

Used to determine behavior when the set value for one or more devices is outside of its low/high limits. If this parameter is set to ALL, the entire request will fail resulting in an exception being thrown and no BDES/VDES values being set for any of the request devices. If this parameter is set to SOME, the set value action will succeed for those set values that are within limits and will not be performed for those set values outside their limits (the returned alarm condition field will be set to TBD). The default setting of this parameter is ALL.

[FUNCTION = { NOFUNC | TRIM | other }]

Specifies whether a trim operation (or some other institution specific operation) will be performed. If NOFUNC, no operation will be performed after setting the named attributes.

Data Type Definition

1. string[]: EPICS device name
2. double[]: attribute field value

In the magent service, the attribute field value will usually mean the B field value.
The above data type may be used for magnet readback or control.

Examples

Example 1: Get the stepping motor vdes property of all the stepping motors in the named beamline ("lcls"):

```
> pvget LCLS/STEP:VDES
STEP:DR12:1       0.45244443
STEP:DR12:2       0.45234376
STEP:DR12:3       0.745568
STEP:DR12:4        0.73079723
STEP:DR12:10       4.106982
STEP:DR13:164         18.1
STEP:DR13:364         58.0
STEP:LI02:119         18.8
STEP:LI02:125         16.3
STEP:LI02:133         16.9
```

```
...
```

Example 2: Get all the y corrector bdess in the “lcls” beamline

```
> pvget LCLS/YCOR:BDES
 YCOR:LI02:118   0.0063584098
  YCOR:LI02:124   6.4847135E-4
  YCOR:LI02:130   -0.008676844
  YCOR:LI02:139    0.008737905
  YCOR:LI02:202   -0.009077272
  YCOR:LI02:212    0.007374592
  YCOR:LI02:222  -0.0017992903
  YCOR:LI02:232   -9.634675E-4
  YCOR:LI02:302    0.006188033
  YCOR:LI02:312   -0.004611565
  ...
```

Examples 3 and 4 are examples of how set would look from the command line, but clearly this would be more common from java, c or matlab;

Example 3: Put the values of all quads in lcls, and trim, taking values from a file:

```
> cat fileofmagetvalues | pvset lcls/ycor:bdes -o FUNC=TRIM
```

Example 4: Put the values from file of named magnets and values, and trim. Don’t abort if any fail to trim:

```
> cat fileofmagnetnamesandvalues | pvset magset/bdes -o FUNC=TRIM -o CONDITION=SOME
```

Klystron Service

The klystron service allows clients to control or readback the activation or deactivation of a given klystron on a given “beam” (described below), and to get/set key operational parameters of a klystron[4].

The precise semantics of the “beam” parameter will vary by institution, but at its core, it is the name of a timing/event definition. In turn, a “timing definition” defines, for a set of beamline devices (possibly all device in the machine), when they must operate with respect to some laboratory timing signal like a copper wire timing fiducial. The name of such a timing definition might be called the “beam identifier”.

Use Cases

Klystron profile activation, energy management and optmization, phase and amplitude feedbacks.

Interface Summary

Get

1) The activation/deactivation status of a given klystron on a given beam.
2) The present phase and/or amplitude of the klystron.

Property	Options	Option Value	Description	Return
active	BEAM	unspecified	Is klystron active on beam	active: boolean
Phase	n/a	n/a	Get klystron phase	The phase readback: double

Syntax:

Get by named beamline:

```
get <beamlineidentifier>/<device-type>:<attribute>
eg double pvget lcls/klys:pdes
```

Get by list of names:

```
KLYSGET/<attribute>
eg cat fileofklsynames.txt | pvget KLYSGET/active -o BEAM=1
```

Set

1) Activate/deactivate a given klystron on a given beam, returning boolean whether the activation was successful.

[4] The name “klystron” is used for simplicity but should be taken to mean generally an RF delivering device whose operation is dependent on beam pulse timing.

2) Set the desired phase or amplitude of the klystron, returning the resulting phase. The phase attribute name, units (degress or radians), and precise interpretation of the phase, probably can't sensibly be standardized).

Options:

[EXECUTE = YES | NO] Whether to execute the change request immediately, and perform follow up actions.

Setting properties of a given klystron, on a given beam:

Property	Property Value	Option	Option Value	Description	Return
active	1	BEAM	unspecified*	Activate klystron on given beam	boolean - Whether activation was successful
	0	BEAM	unspecified*	Deactivate klystron on given beam	boolean - Whether deactivation was successful
Phase	float	EXECUTE (optional)	boolean	Set klystron phase, and optionally whether to execute this phase change request immediately	float - The phase readback

Syntax:

Set by named beamline:

```
get <beamlineidentifier>/<device-type>:<attribute>
eg double pvget lcls/klys:pdes -o TRIM=1 23.34
```

Set by list of names:

```
KLYSSET/<attribute>
eg cat fileofklsynamesandactiveflags.txt | pvget KLYSSET/ active -o BEAM=2
```

Data Type Definition

1. string[]: EPICS device name
2. double[]: attribute field value

Examples

Example 1: Get the activation status of all klystrons in beamline group “linac”, on timing beam identifier 10:

```
$ pvget linac/klys:act -o beam=10
KLYS:LI20:21  1
KLYS:LI20:31  1
KLYS:LI20:41  0
KLYS:LI20:51  1
KLYS:LI21:21  1
KLYS:LI22:21  1
KLYS:LI22:31  0
KLYS:LI22:41  1
KLYS:LI23:21  1
KLYS:LI23:21  1
...
```

Example 2: Set the phase of 2 named klystrons, and execute the phase setting. This example uses a unix “here” file to give the names and values right in the bash command line:

```
$ pvset klysset/phas -o execute=yes <<EOF
KLYS:LI20:21  23.34
KLYS:LI20:31  45.45
EOF
```

Archive/History Service

Gets the archived data points for a given device property, over a given long term interval (seconds to years).

Use Cases

General beam and operational diagnostics, e.g. investigating whether an accidentally reset bend magnet is causing a beam oscillation.

Interface Summary

Gets:

Gets archived data points for a given device property

Options:

STARTTIME = ”mm/dd/yyyy hh:mm:ss”[5]

[ENDTIME = ”mm/dd/yyyy hh:mm:ss”] (default is now)

[DATEFORMAT = DATEFORM0 | MMDDYYYY | MMDDYYYY_FRAC | SDN]

The format of the data time strings returned. DATEFORM0 is the default format: "dd-mmm-yyyy hh:mm:ss" - which is Matlab dateform number 0. MMDDYYYY is the format "mm/dd/yyyy hh:mm:ss". MMDDYYYY_FRAC is the format "mm/dd/yyyy hh:mm:ss.hh". SDN is short for Serial Date Number, used in matlab, and defined as “the whole and fractional number of days from 1-Jan-0000 to a specific date.”

Sets:

No requirement. The archive data store will have its own backend interface independent of the client side API.

Returned Data Definition:

The archive data is given as a table of timestamped values, containing a rectangular matrix of 5 arrays:

i) double[]: the values (or double [][] for waveform)

ii) <dateform>[] times: (in the date format specified in the DATEFORM option)

[5] For this absolute basic definition of interfaces, it’s probably reasonable to define that the Archive interface must accept at least STARTTIME and ENDTIME given in mm/dd/yyyy hh:mm:ss format. However, implementations should additionally support STARTTIME and ENDTIME with relative inputs like “yesterday”,”last week”, “10 days ago”; see unix date command --date option for possibilities.

iii) int[]: repeat count. May well be valued 0 for all data (no repeats)

iv) int[]: the pulse id of the data (or head data in case of waveform)

v) int[]: data count. 1 for a scalar and greater than 1 for a waveform-- count may be used to index into the values array for waveforms to extract the waveform values for each sample

vi) int[]: a flag indicating whether the times represent Daylight Savings Time.

Examples

Example 1. Get the machine rate over the last day:

```
$ starttime=`date --date="1 day ago" +"%m/%d/%Y %T"`

$ now=`date +"%m/%d/%Y %T"`

$ pvget IOC:IN20:MC01:LCLSBEAMRATE/HIST.lcls -o STARTTIME="${starttime}" \
  -o ENDTIME="${now}"
  60.0  24-Sep-2010 20:53:00  0   973  1  1
  60.0  25-Sep-2010 15:29:07  0  7005  1  1
  60.0  25-Sep-2010 15:29:09  0  2392  1  1
  24.0  25-Sep-2010 21:51:05  0  8191  1  1
   0.0  25-Sep-2010 21:51:07  0  8199  1  1
```

In example 1, the data source was being monitored by the archiver, and only changed 5 times in a day (machine was down).

Assumptions, Definitions, Q & A

At each facility a naming convention will map EPICS PVs to beamline devices (say the first 3 fields of the EPICS PV name identify the device), and some other field identifies a attribute of the device. Eg In PV name "XCOR:LI21:245:BDES," XCOR:LI21:245 identifies a device, and BDES identifies a property.

At each facility there will be some beam timing identifier. I don’t think we can standardize this.

Define an element to mean a modelled element in the MAD sense. An element may correspond to an EPICS device, but not necessarily. Eg Mad Element "X245" may be EPICS device "XCOR:LI21:245". However, there may well be elements that have no corresponding EPICS device, eg "MARK4" might be a modelled marker point, for which no EPICS PV is needed. *So, don't expect all rows in a table of all the R-matrices of an accelerator to map to a PV, and for those that do, don't expect only 1 row to match a single PV*.

<u>Q & A</u>

Q: What you’ve described look like APIs. Why aren’t these services just implemented as APIs of shared libraries or jar files rather than remotely accessed services?

A: Because client types for scientific data are diverse: matlab, mathematica, Java programs, C/C++, python, jython, web application servers and their clients. for each one you supported, you’d have to separately implement the backend connections to device channels, persistent stores, oracle, etc.

References

[1] Accelerator Independent Data Access (AIDA) [1], http://www.slac.stanford.edu/grp/cd/soft/aida/. See in particular links under the section "Individual Service Data Users Guides."

[2] AIDA SLC Bpm Data Provider Guide http://www.slac.stanford.edu/grp/cd/soft/aida/slcBpmDpGuide.html

[3] AIDA EPICS Channel Archiver Data Provider Guide http://www.slac.stanford.edu/grp/cd/soft/aida/epicsChannelArchiverDpGuide.html