Using Data Access, First Impressions

Kay Kasemir

October 2005

Disclaimer

- This is about work in progress.
  The DataAccess interface as presented in here has not been released, nor is it in its final format.
  DataAccess is available as a CVS Snapshot on
  http://www.aps.anl.gov/epics
  and it's neither guaranteed to compile
  nor to do anything useful, yet.

V4, Channel Access, Central Role of Data Access

- A significant portion of the V4 novelties is in Channel Access
  - More user control over subscriptions:
    Per-client rate, max rate, custom events, …
  - Custom data containers:
    No longer limited to predefined list of DBR_… types.
- "DataAccess" is proposal for interfacing these arbitrary data containers between data sources and destinations.
- Sources include
  - Program "writing" data via the CA client library.
  - CA client library callback presenting the result of a "read" request.
  - Program (server tool) serving data via the CA server library.
- Destinations may reside in
  - CA client lib. Trying to "write" the data.
  - User code parsing the "read" result.
  - CA server library reading the server tool's data.
- Unclear right now if other EPICS APIs like record or device support could also use "DataAccess".

Example Source and Destination Data

- ```
  struct source
  {
    time   timestamp;
    double value;
    string units;
    enum   alarm_severity;
    struct color
    {
     int red, green, blue;
    }
    int    pulse_type;
  }
  ```

- ```
  struct destination
  {
    time   timestamp;
    float  value;
    string units;
    enum   alarm_severity;
    int    color_table_idx;
  }
  ```

- Properties timestamp, value, …
- Types time, double, string, …

- Properties may differ
- Types for matching properties may differ

Data Access Ideas

- Interface
  - Does not hold any data (as e.g. ‘GDD’ used to do)
- Generic
  - User should not have to arrange data in any special way; there is no common “EpicsDataObject” from which one must derive.
  - Instead add implementation of DataAccess “PropertyCatalog” interface.
- Properties
  - Defined as strings/names “value”, “units”, …,
    converted into numeric IDs for performance.
  - Need mutual agreement on names.
    No magic mapping from e.g. “color” to “color_table_index”.
- Types
  - Data Access tries to convert *double* into *float* etc.
    Need to define if string “42.5 Apples” ought to convert into int 42.

DataAccess PropertyCatalog

- The one and only interface one has to implement and understand for using DataAccess.
- Designed in C++, but ideas should work in Java, Perl, … as well.
- Source and destination must both implement PropertyCatalog
- Given

```
PropertyCatalog &src, &dest;
```

  one can do this:

```
assign(dest, src);
```

  to copy all matching properties from source to destination, converting types as necessary.

- Interface PropertyCatalog:

```
status traverse (propertyViewer &v);
bool find (propertyId & id, propertyViewer & v);
```

PropertyCatalog::traverse

- DataAccess will invoke traverse to visit the data.
  Need to ‘reveal’ all properties via their ID.

```
static const propertyId value_id(“value”);
…
status traverse (propertyViewer &v)
{
    v.reveal(value_id, source.value);
    …
    return OK;
}
```

- There are actually variants of traverse to support
  - Read-only, viewing traversal
  - Writing traversal
  - Traversal of only property & type information, no data.

PropertyCatalog::find

- Used by callers to locate a specific property without traversing the whole PropertyCatalog

```
bool find (propertyId &id, propertyViewer &v)
{
    if (id == value_id)
    {     v.reveal(value_id, source.value);
          return true;
    }
    else if (id == …
    …
    return false; /* unknown property */
}
```

- There is a “locator” helper class for registering reveal methods, keeping them in a hash based on the property ID, to avoid the chain of “if (id == …) …”.

Property Viewers

- The find() and traverse() methods reveal data items, and the Property Viewer needs to handle every data type:

```
class propertyViewer
{
  virtual void reveal(propertyId &, double &);
  virtual void reveal(propertyId &, int &);
  …
}
```

- There is a …Viewer for static data, …Manipulator for write access, and maybe a new variant for type information

(My) Misconception

- Given a propertyCatalog *pc from e.g. a CA read response, there is no querying/pulling interface like this:

```
double value = pc->getProperty("value")->toDouble();
cout << "The value is " << value << endl;
```

- Also no iterating interface like this:

```
foreach property (  pc->getProperties() )
{
   print property->getName(), " is ",
         property->toString();
}
```

- Instead, invoking

```
pc->find(value_id, my_viewer);
```

or

```
pc->traverse(my_viewer);
```

will transfer the program flow to data access, which will then call the reveal methods inside "my_viewer" at its discretion.

What already works

- Possible to define PropertyCatalogs for structs of double, int, ..
- Assignment of property catalogs to each other to matching properties is trivial.

```
propertyCatalog *source, *dest;
assign(*dest, *source);
```

- One can write a data-copying viewer to be used like this:

```
// All the viewer's reveal() methods
// copy data into instance variable "double data"
DoubleViewer v;
pc->find(value_id, v);
cout << "The value is " << v.data << endl;
```

to effectively get a "pulling" interface for known properties.

What I didn't accomplish

- Strings
  - Accessed via "stringSegment",
    defined in "daString.h" and "daStream.h",
    but the latter doesn't get installed?
- Hierarchy
  - Assume the source catalog has more than one "color":

```
source.display.color
source.beam.color
```

  - With a "pulling" interface, one could pick a specific one like this:

```
pc->getProperty("display")
     ->getProperty("color")
        ->getProperty("red")->toInt();
```

  - Unclear which of the "color" properties `assign()` would pick, so one needs to implement a `propertyViewer` with a state machine that tracks the callback path.

What I didn't accomplish..

- Type Info
  - Remember that if a PropertyCatalog containing a "double value" is subjected to find() or traverse(), the propertyViewer's
    ```
    reveal(propertyId &value_id, double &data);
    ```
    is invoked.
  - This currently requires an actual instance of the data.
  - To learn if there is a 'value' property and what it's type is, one needs to invoke
    find(value_id, viewer);
    and then take notes inside the viewer which overloaded reveal was called. Doable, but necessary?

Issues under Discussion

- Types
  - Is this the supported list?
    Octet, bool, int16, 32, 64, float32, 64, string, enum, time.
  - Structures and arrays of the above.
  - Support unsigned integers?
- String and array interfaces
  - Currently written in order to support segmented storage.
    Consequence:
    Cannot access string as (const char *),
    only allowing char-by-char callbacks getChar/putChar.
    Andrew Johnson proposes a “StringReader/Writer” API that is closer to the familiar std::string, MFC::CStr, …