Data Access Update

Jeff Hill

What is This – An Interface

- A plug-compatible interface
  - Introspecting arbitrarily complex data
  - Data Exporter
    - Intimately familiar with proprietary data
    - Facilitating access from generic programs
      - Providing functions conforming to the interface
  - Data Client
    - A generic consumer of data
    - No preconceiving knowledge of the data
    - Introspects the data
      - Calling functions in the interface

What is This – A Support Library

- A support library
  - Equivalence and copy operator for dissimilar data sets
    - Leveraging the well defined interface

Why We Need It – Expanded Meta Data

- Current EPICS has a fixed set of meta-data
  - This obviously needs to expand
  - EPICS base developers don't anticipate
    - All possible meta-data
    - All possible meta-data permutations
  - Application developers define new meta-data
    - Expansion of toolset will hopefully accelerate
- Decentralized development requires proper decoupling of components from each other
  - Changes in one component do not cause another component to break
- Data Access is about expanding this set while keeping the tools properly decoupled
  - If the meta-data is expanded in a data source
    - We need not rewrite all of the clients of that source
    - Support library efficiently extracts a subset

Why We Need It – Multi-Parameter Synchronization

- In multi-agent systems synchronization is a reoccurring theme
- Current EPICS synchronizes single parameter with a fixed set of meta data
- Data Access is about synchronizing arbitrary application defined data capsules with
  - Time (a time stamp)
  - An arbitrary (application defined) event
    - RF arc-down event etc (data acquisition)
  - A client's read or write request
    - Synchronized multi-channel read / write

Why We Need It – Device Orientation

- Intelligent instruments are the norm
- Intelligent devices require message passing
  - Essential for tool based approach to devices
- Devices *must* defined arbitrary request / response capsules
  - Data Access interfaces this arbitrary data capsule

Why We Need It – Device Orientation

- Device record
  - Goal:
    - Device level abstractions w/o writing low level code
  - Three components
    - Interface
      - What multi-parameter messages are accepted and what multi-parameter responses are sent
      - What events are posted containing what properties
    - Behavior
      - Probably state notation language
    - Data
      - Probably other records
  - New features in iocCore's dbCommon

General Design Philosophy

- User isn't required to store his data in a particular format
- Nevertheless, knowledge of the structure of the data determined at compile time
  - Access to the data can be efficient.

General Design Philosophy

- This is not office computing
  - Designed for use in limited memory embedded systems
- Data Access interface must not preclude user data stored in multiple non-contiguous blocks
  - Free lists based memory allocation
    - Low latency
    - No fragmentation
    - Predictable behavior

General Design Philosophy

- Data Access interface must not require C-RTL general purpose memory management AKA malloc
  - Passing data via data access in high throughput situation
    - Efficiency gets noticed
    - Data access interface to application data lifetime is duration of a function call
      - Malloc is a very high overhead call in this context

Interface Details - Properties

- **All Exported Data Assigned a Property Name**
  - “weight”, “units”, “maximum”
    - Any name that a group of programs mutually agree upon
  - Properties may have subordinate properties
    - “value” property
      - “units” subordinate property
      - “high limit” subordinate property
      - “low limit” subordinate property
- Property id’s are stored in type daPropertyId
  - Class constructor requires a property name string

Interface Details – Writing a Data Exporter

- **Data Exporter derives from class daData**
- **This means nothing more or less than**
  - **4 functions provided by the Data Exporter**
    - Traverse, writeable
    - Traverse, readable
    - Find, readable
    - Find Writable
  - These functions are called by the Data Client when it introspects the data

Interface Details – Writing a Traverse Function

- Data Exporter's traverse function has one incoming argument
  - Reference to data publishing adaptor
- Data Exporter calls a function in this adaptor for each exported property
  - Adaptor has overloaded functions
    - One for each primitive type
  - No write access – readAdaptor::reveal called
  - Write access – writeAdaptor::expose called

Interface Details – Writing a Traverse Function

```
void myData::traverse ( readAdaptor &adt ) const
{
    adt.reveal ( propertyHighDisplayLimit, data.hdl );
    adt.reveal ( propertyLowDisplayLimit, data.ldl );
    adt.reveal ( propertyWeight, data.weight );
    adt.reveal ( propertyHeight, data.height );
}
```

Interface Details – Writing a Find Function

- Called by Data Client
  - To index a property by its property identifier
- Compared to the traversal mechanism …
  - Additional flexibility
  - Some well bounded loss of runtime efficiency
- Data Exporter's find function is passed a data publishing adaptor and a property id
  - Choice of indexing method left to exporter
    - Prototype in support libraries

Interface Details – Writing a Find Function

```
void myData::find (
        const daPropertyId & id, readAdaptor & adt ) const
{
    // efficent approach for when there
    // are more properties implenmented in
    // support libraries
    if ( id  == propertyHighDisplayLimit ) {
        adt.reveal ( id , data.hdl );
    }
    else if ( id  == propertyLowDisplayLimit ) {
       adt.reveal ( id , data.hdl );
    }
}
```

Interface details – Subordinate Properties

- If there are subordinate properties
  - the reveal / expose function are supplied with an optional 3rd argument
  - A reference to type daData
- Recall that the Data Exporter derives from class daData
  - This 3rd argument references a Data Exporter for the subordinate properties

Interface Details – Writing an Array Data Exporter

- **Array Data Exporters derive from class daArray**
- **This means nothing more or less than**
  - **8 functions provided by the Array Data Exporter**
    - Traverse array, writeable & readable versions
    - Traverse multidimensional array slice, writeable & readable versions
    - Number of dimensions, get & set versions
    - Dimension bound, get & set versions
  - These functions are called by the Data Client when it introspects array data
- One of the overloaded functions in the scalar publishing adaptors has type daArray

Interface Details – Writing an Array Traverse Function

```
void myArrayData::traverse ( readArrayAdaptor &adt ) const
{
    // arrays may be stored in non-contiguos blocks
    // multidimensional arrays are revealed in
    //
    // multidimensional arrays revealed in row-major
    // order followinhg convention for the C language
    //
    adt.reveal ( propertyValue, data.arrayChunk0, 256 );
    adt.reveal ( propertyValue, data.arrayChunk1, 256 );
}
```

Interface Details – Enumerated Types

- String Exporter derives from class daEnum
- Exporter supplies these functions
  - Get number of states
  - Traverse states
  - Convert state string to int
  - Convert int to state string
  - State is valid test
  - Remove state, set string for state
- Any primitive type convertible to C type “int” may store the state
- One of the overloaded function in the publishing adaptors has type daEnum

Interface Details - Strings

- More complicated than expected!
- Some requirements
  - No raw access to the character string
    - Strings may be stored in non–contiguous blocks
      - Many C-RTL things such as scanf don't like this
      - The class std::string doesn't allow this
  - String exporter must not be forced to call malloc in its constructor
    - Most std::streambuf implementations do
    - Most std::string implementations do
  - Support for wide strings is desirable
  - Don't pass off string to numeric conversion to the string storage implementation?
    - Want consistent approach when converting strings to numbers
    - Many numeric types – best to avoid a fat interface

Interface Details - Strings

- String Exporter derives from class daStringIO
- Exporter supplies these functions
  - Get a character – unsigned int passed out
  - Put a character – unsigned int passed in
  - Put string – daStringIO reference passed in
    - Facilitates high speed copy
  - Get std::locale reference
- One of the overloaded functions in the scalar publishing adaptors has type daString

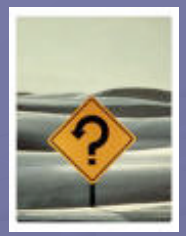
Interface Details - Time Stamps

- Data access design philosophy
  - Don’t stipulate the primitive storage type
    - epicsTime is versatile, but should not be stipulated
- Therefore, we need a daTime interface that time stamp exporters derive from

Frequently Asked Questions

- Whoa, this thing is called Data Access!
  - Don't O.O. systems use messages and remote procedure calls?
    - Public data is anathema
- Data Access was invented for the purpose of passing messages
  - To specify the parameters of the messages, and map between dissimilar messages

Frequently Asked Questions



- Why not use a conventional RPC system like CORBA?

- Issues with RPC based distributed architecture
  - CORBA is a low level system
    - Unconstrained use could lead to spaghetti distributed architecture
  - Connection management (lack thereof)
    - Difficulty predicting system degradation if one node is lost
    - System startup chicken and egg problems
  - Proper integration into embedded and preemptive scheduled OS
    - Proper system degradation under load
    - Memory management in embedded systems
  - Vendor uniformity and lifetime

- Streaming message transport systems
  - This is different from conventional RPC systems
    - RPCs typically require a network round trip for each message
  - This is what makes high throughput possible

Alternative Approach

- Why not use a data description compiler like IDL or XDR?

- This is certainly worthy of consideration, but …
  - Proper decoupling of sender and receiver data spaces appears to be important for a tool based approach
  - Conventional data description compiler based systems require interfaces of the sender and receiver be identical
    - Parameter-for-parameter, field-for-field, bit-for-bit
    - Sender and Receiver must have the same repository ID
    - If not, no communication
  - An event may have many associated properties
    - Clients will rarely need all of them, and there will be many permutated subsets

Common Misconceptions

- ✘ This is a C++ template based interface
  - ☑ In fact, pure virtual base class based interfaced
  - ☑ Templates are used only in the implementation of the support library
    - ☑ Templates not seen by users
- ✘ This is a data object
  - ☑ In fact, a universal interface to non-uniform data
    - Proprietary data storage formats need not change
    - We are not designing a class that allocates space for, enforces a storage format for data
      - Memory isn't allocated by the library
        - for arrays, strings, containers etc
      - This isn't GDD or cdevData

Common Misconceptions

- This interface isn't compatible with C, or Java, python, …
  - C++ Data Exporter can access data maintained by C programs
  - All of the interfaces described here could have C, java, python … wrappers
  - A pure Java implementation could be written
    - No templates in Java, but when creating the support library a program that creates a program could be written as was done with GDD and its ancestors

Recent Changes

- Interface to arrays has been greatly simplified
  - No strides, chunks etc
  - No arrayXActionContext, arrayRequest
- Property hierarchies - after careful thought
  - Every property might potentially have subordinate properties
  - Allows for proper evolution of structured data
    - If it's a scalar w/o limits today then it should not be forced to become a container to have limits tomorrow
- String converter class not passed into every reveal / expose function
  - Enumerated types interfaced through class daEnum
  - Not all primitive types can be enumerated
    - Must be convertible to primitive type "int"
- Use of member templates
  - Simplifies support libraries
    - Reduces use of macros

Conclusion

- Data Access – a key facilitator for EPICS upgrades …
  - Expanded meta data set
  - Multi-parameter synchronization
  - Data Acquisition
  - Device Orientation
  - Tool based approach requires proper decoupling of tools from each other
    - changes in one tool do not cause another tool to break
- There are simple steps involved in writing a data exporter