DØ Note XXXX

Proposal for a Persistent Object Manager using DSPACK with ODMG C++ Interface

H. Greenlee, J. Hobbs, S. Snyder, J. Womersley

December 9, 1996

This document contains a proposal on how to make a C++ interface to DSPACK [1] that partially conforms to the ODMG C++ binding standard for object oriented databases [2]. Because of the limitations of DSPACK, a full implementation of the ODMG standard is not possible. By adhering to the restrictions required by a partial implementation of the ODMG standard, it is hoped that a future migration of DSPACK programs to use a true object oriented database could be accomplished relatively easily.

1 Overview of DSPACK

DSPACK is a data handling package that allows objects to be defined and manipulated at execution time. DSPACK also provides input/output capability for its objects. This section provides a brief summary of the main features of DSPACK.

1.1 Object Model

The basic entity of the DSPACK data model is called a *data set*. A data set is a linear contiguous array of fixed length objects called *data set entries*. The number of entries in a data set is variable at execution time.

1.1.1 User-defined types

A data set entry is a fixed length user-defined type that consists of an arbitrary collection of predefined atomic data types, user-defined data types, fixed length arrays, and pointers. A data set entry can be thought of as an arbitrary C struct.

1.1.2 Pointers

Data set entries can contain pointers to other data set entries, which need not be in the same data set. In the current version of DSPACK the type of data set entry that is pointed to must be specified at object definition time. That is, DSPACK currently lacks the concept of run time binding. A partial implementation of run time binding may be available in future versions of DSPACK.

1.1.3 Names

Each data set has a unique 16-character name. Thus DSPACK is not friendly to data models that require indefinite replication identical data sets. DSPACK prefers to achieve indefinite replication by multiplying entries within data sets. DSPACK prefers that data sets be located by name rather than by pointer. DSPACK returns to the user a handle (a pointer to a pointer) the first time a data set is located. Thereafter, the data set can be referenced by handle for more efficient access. The validity of handles and pointers embedded in DSPACK data sets is maintained accross object relocations.

1.2 Object Definition

DSPACK always allows data sets to be defined at execution time via subroutine arguments or via object description (.d) files. DSPACK description files are similar to C or C++ header files for defining structs. Here is an example of a .d file for defining an object similar to the DØ HEAD Zebra bank:

```
DEFINE char8 C*8 0
       "Eight character string" string
END DEFINE

DEFINE head_t S 1
        .INT4   record_type
        char8   microname       ! Level 2 node (6 char.).
        .INT4   date_time(2)    ! VAX 64-bit time.
        .INT4   local_run       ! Local run number.
        .INT4   input_event(2)  ! Crossing number (16/24 bits).
        .INT4   event           ! Output event number (data logger).
        .INT4   event_type      ! Assigned in level 2 or Monte Carlo.
        .INT4   trigger_bits    ! Level 1 bit mask.
        .INT4   run             ! Global run number
        .INT4   code_version
        .INT4   header_version
        .INT4   filter_bits(4)  ! Level 2 bit mask.
        .INT4   reco_bits(10)   ! Reco bit mask.
        .INT4   error_code      ! Level 2 error code.
        .INT4   flags           ! Bit 0 = microblank
        .INT4   omni_bits(2)    ! Omni-filter bits.
END DEFINE
```

DSPACK description files are readily converted to C or fortran header files. DSPACK provides a utility to do so. Except for characters, DSPACK does not supports atomic data types shorter than 32 bits.

2 Overview of ODMG C++ Binding

The section describes the basic features of the ODMG standard and compares them to equivalent features of DSPACK.

2.1 Object Model

The ODMG standard distinguishes between transient and persistent (actually persistence capable) classes. A transient object is an instance of an arbitrary C++ class that exists only in memory for a time not exceeding the lifetime of the program. Persistent classes can be instantiated as either transient objects or persistent objects associated with a particular database. Persistent classes must conform to the ODMG object model, which restricts the types and classes of which they can be composed.

2.1.1 User-defined types

ODMG persistent classes can be built up from predefined atomic data types, user-defined data types, smart pointers, fixed length arrays, and a predefined set collection objects. The ODMG standard supports the usual list of atomic data types (integer, float, character, etc.). The set of atomic allowed by ODMG and DSPACK is similar except that DSPACK has a built in 32-bit word length.

2.1.2 Collections

The ODMG standard supports the following parameterized collection objects:

d_Collection<T>	Generic collection (abstract class)
d_Set<T>	Unordered collection, no duplicates
d_Bag<T>	Unorderered collection, duplicates allowed
d_List<T>	Ordered collection
d_Varray<T>	Variable length array
d_Iterator<T>	Generic iterator for any ODMG collection

The built-in C array type ([]) can also be used to construct fixed length arrays. Note that persistent classes can not be built out of arbitrary classes whether parameterized or not.

In contrast, DSPACK offers only fixed length arrays and the single type of collection represented by the data set.

2.1.3 Pointers and Relationships

Pointers embedded in persistent objects are required by be one of a number of predefined smart pointer types. All of the smart pointer types overload the dereference operators * and ->. Here is a summary of the smart pointer types:

d_Ref_Any	Unidirectional pointer to anything
d_Ref<T>	Unidirectional pointer to T
d_Rel_Ref<T,M>	One-to-one relationship with T
d_Rel_Set<T,M>	One-to-many relationship with T
d_Rel_Bag<T,M>	One-to-many relationship with T
d_Rel_List<T,M>	One-to-many relationship with T
d_Rel_Varray<T,M>	One-to-many relationship with T

The interface of the one-to-one relationship class d_Rel_Ref<T,M> is consistent with d_Ref<T>. The interface of a one-to-many relationship class such as d_Rel_Set<T,M> is consistent with d_Set<d_Ref<T>>. Relationships differ from simple unidirectional pointers in that they are bidirectional. Relationship pointers always come in pairs. The run time system guarantees the consistency of the the relationship whenever either side is modified.

DSPACK currently has no concept of either smart pointers or bidirectional relationships, although we have been told of plans for DSPACK to support disbributed data files. Implementing any of the smart pointer types in a non-trival way therefore would require some effort, but such pointers will be very useful when the time comes to store events in a disbributed fashion.

2.1.4 Names

The ODMG standard allows, but does not require, objects to have one or more names. ODMG names must be unique within a single database (unlike DSPACK which only requires names to be unique within one event). The ODMG standard does not mention any maximum limit on the length of names.

2.1.5 Inheritance and polymorphism

The ODMG standard supports single and multiple inheritance and polymorphism. DSPACK currently does not support polymorphism.

2.2 Object Definition

ODMG persistent classes must inherit from a root persistent object class called d_Object. Depending on implementation, the data content of persistent classes is defined using a standard language independent Object Definition Language (ODL), directly as C++ headers, or using some other data definition languate. Whatever language is used, additional object interface and implementation needed to implement persistence is added by an object definition preprocessor. The output of the object definition preprocessor consists of C++ header and source files.

2.3 ODMG C++ classes

The ODMG standard defines several classes that can be used to perform database operations and manipulate persistent objects. Some low-level ODMG classes, including collections and smart pointers have already been discussed in Sec. 2.1. In this section we describe the few remaining classes defined by the ODMG standard that control higher level database operations. Here is a summary.

d_Error	Generic exceptions (abstract class)
d_Database	Database object
d_Transaction	Database transaction object
d_Object	Generic persistent object (abstract class)

2.3.1 Class d_Error

Instances of d_Error are thrown in response to various execution time error conditions.

2.3.2 Class d_Database

The class d_Database is a transient class whose instances are associated with a particular physical database. This class contains methods to open and close databases (called open and close) and methods for doing name-based operations with persistent objects (set_object_name, lookup_object, and rename_object). ODMG names differ from DSPACK names in that the scope of their uniqueness is larger (the whole database compared to just one event).

2.3.3 Class d_Transaction

The ODMG standard requires all operations on persistent objects to be grouped into transactions. The transaction class has four methods with the obvious names begin, commit, abort, and checkpoint. From the point of view of the outside world, transacions are atomic, becoming visible only after a commit operation. The abort operation reverts the database, and any persistent objects in memory, to their states prior to the last begin.

DSPACK of course knows nothing of transactions and the associated complications of record locking and concurrency, having instead an event oriented I/O interface.

2.3.4 Class d_Object

All persistent classes inherit from the base class d_Object. The d_Object class defines the following three nontrival groups of methods that are not found in transient classes.

1. Overloaded new and delete.

2. Mark_modified.

3. D_activate and d_deactivate.

The purpose of these methods is summarized below.

New and delete The ODMG standard requires that each persistent class have three overloaded versions of the new operator. One version is used to create transient instances and two versions are used to create persistent instances of a persistent class. The status of an instance as transient or persistent can not be changed after it is created. The prototypes of the overloaded new and delete operators are as follows:

```
class d_Object {
public:
    void* operator new(size_t, size);          // Transient instance
    void* operator new(size_t, size,           // Persistent instance
                       d_Database *database,
                       const char *typename);
    void* operator new(size_t, size,           // Persistent instance
                       const d_Ref_Any &clustering,
                       const char *typename);
    void operator delete(void*);
```

The first form of the new operator is equivalent to the default new operator. It creates a transient instance of a persistent class. The last two forms of the new operator create a persistent instance of a persistent class by associating the instance with a particular database or with another persistent object. The third argument of the persistent forms of the new operator is a character string that is used to look up the definition of the object in the database schema.

The delete operator deletes an object from memory and from the database if it is persistent.

Mark_modified The mark_modified method sets a status flag to indicate that a persistent object has been modified. The database run time system tests this flag to determine which persistent objects need to be written to disk. The mark_modified method must be called whenever a persistent object is modified. This call would normally be done automatically within any methods that modify the state information of an object.

In DSPACK the same functionality is provided by the FSMARK subroutine. Only marked data sets are written to disk.

DSPACK and ODMG both lack the concept of a global write that unconditionally writes all persistent objects to disk. In a program that reads and writes events sequentially, a way must be provided to ensure that all required objects are marked for output.

D_activate and d_deactivate The d_activate and d_deactivate methods are called by the run time system when an object is read into or removed from memory (even though it still exists in the database).

3 Summary of DSPACK and ODMG usages

This section summarizes and compares some equivalent DSPACK and ODMG usages for various common persistent object operations.

3.1 DSPACK Usage

System initialization	call ds_init(server_name, ierr)
Open file for input	call fsopin(ioid, file, chopt, ierr)
Open file for output	call fsopou(ioid, file, chopt, ierr)
Read an event	call fdsin(ioid, 1, ierr)
Find a data set by name	call dsnivr(name, handle, ierr)
Create a new data set	call dsbook(name, handle, maxent, ierr)
Expand an existing data set	pnt = idsadd(name, handle, ient, ngot, isize, nentot, ierr)
Mark data set for output	call fsmark(ioid, name, ntot, ierr)
Delete a data set	call dsdrop(name, handle, ierr)
Write an event Close output file	call ds_close_out()

3.2 ODMG C++ Usage

System initialization	d_Database mydb;
	d_Transaction event;
Open file for input	mydb.open(file, read_only);
Open file for output	mydb.open(file, read_write);
Read an event	event.begin();
Find an object by name	obj = mydb.lookup_object(name);
Create a new persistent object	obj = new obj_t(&mydb, typename);
Expand a persistent collection object	obj->insert_element(&element);
Mark object for output	obj->mark_modified();
Delete a persistent object	delete obj;
Write an event	event.commit();
Close file	mydb.close();

4 Design Philosophy

The ODMG standard provides the following five primary types of classes:

1. Persistent object base class d_Object.
2. Collection classes d_Set<T>, etc.
3. Smart pointers and relationships d_Ref<T>, etc.
4. Transaction class d_Transaction.

5. Database class d_Database.

The first three types of classes have to do with the object model. Items four and five have to do with persistent object I/O. We propose that the DØ persistent object mechanism should only implement items 1–3 relatively directly. For event I/O, we propose that an event oriented model is more natural both in terms of its interface and its implementation using DSPACK than the more general transaction oriented model.

In the case of the D0 persistent object model, we propose having a programming interface that does not use the ODMG names directly. We think that this will make it easier for object database I/O (e.g. calibration data) to coexist with event I/O using DSPACK, as it must. The DØ persistent object model will have the following kinds classes.

1. Persistent object base class d0_Object.

2. Collection classes d0_Set<T>, etc.

3. Smart pointers and relationships d0_Ref<T>, etc.

In order to implement I/O to object databases, it will be necessary to convert persistent objects from the DØ persistent classes, based on d0_Object, to ODMG or vendor persistent objects classes, based on d_object. Initially, such conversion would probably involve copying. Likewise, to implement I/O to DSPACK files, it will be necessary to convert DØ persistent objects to DSPACK objects, also initially involving copying. Eventually, it may be possible to develop more efficient conversion methods involving shallow copying or casting.

For the DØ persistent collection classes, it will be more natural to have interfaces resembling the STL collection classes than the ODMG collection classes.

5 DØ Event I/O Interface

For event I/O, we propose that DØ should develop its own classes without reference to the ODMG standard or obect databases. If and when DØ aquires a true object oriented database, we can install its transaction processing underneath the DØ event I/O interface. The DØ I/O classes should be event-oriented in the sense that data is organized into "events" that are not interlinked with other events or non-event objects. Such self-contained event objects are not to be understood as containing exclusively actual event data. For example, they could contain calibration data. Neither physical partitioning of events, nor random access to events is excluded by the interface, although an initial implementation would probably not have either.

The DØ event I/O class hierarchy is shown in Figs. 1–3. The top level class D0Stream is an abstract class containing the I/O interface, meaning basically reading and writing events, and opening and closing files. We assume that there will be a class D0StreamName for storing filenames that behaves in some respects like a string class, but with additional capabilities for searching and parsing filenames. D0StreamName

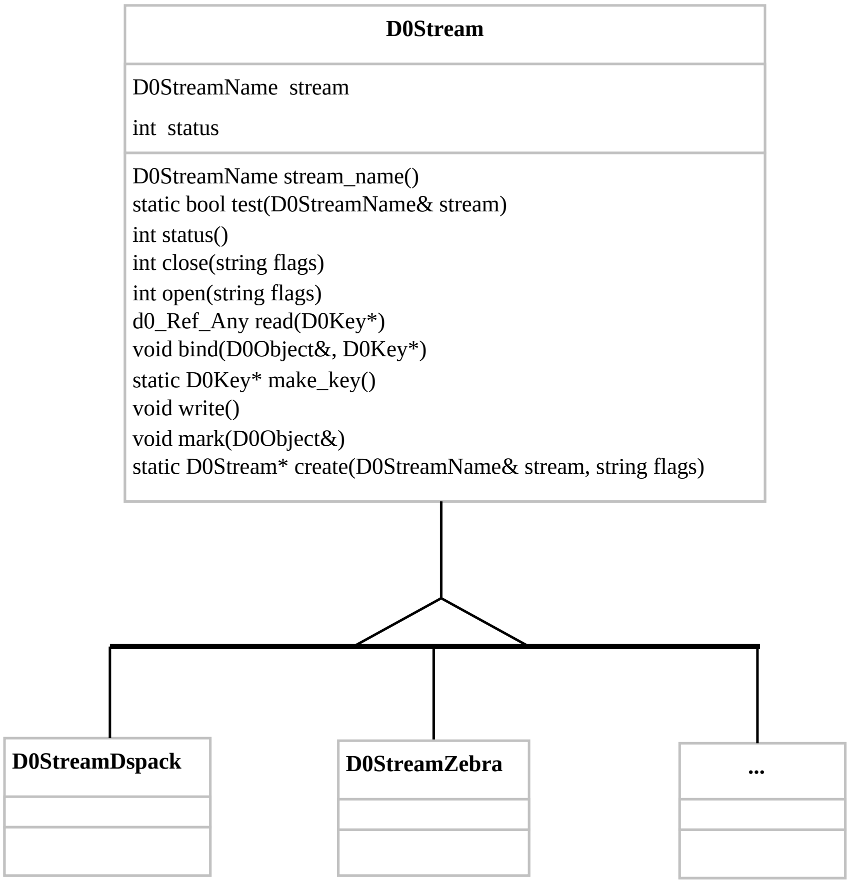

Figure 1: D0Stream class diagram.

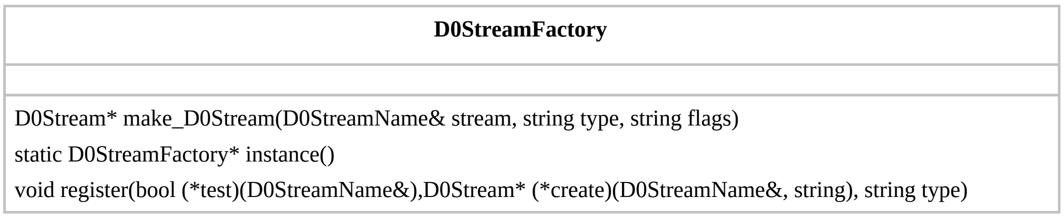

Figure 2: D0StreamFactory class diagram.

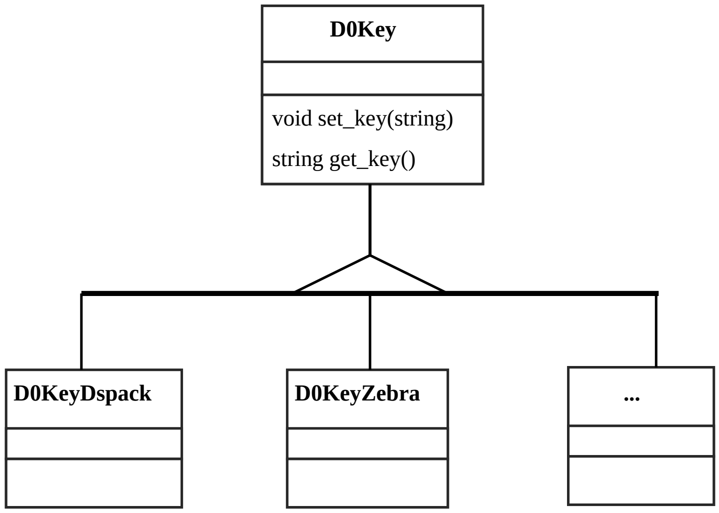

Figure 3: D0Key class diagram.

could also be used to reference more general objects than just disk files, such as wildcard filenames, FATMEN files, virtual streams (D0dad files), shared memory, etc. Below D0Stream are concrete classes corresponding to different physical I/O formats.

5.1 Sequential and Random Access I/O

The event reading and writing interface is intended to be general enough handle both sequential and random access I/O. There is a key class hierarchy (Fig. 3) corresponding to various random access file formats. Purely sequential formats would not need to implement corresponding D0Key subclasses.

The method D0Stream::mark(D0Object&) is used to mark objects for output. The method D0Stream::write would initiate physical I/O for all marked objects. The method D0Stream::bind(D0Object&, D0Key*) is used to associate random access keys with objects. The read method can be used for either sequential (with null key) or random access I/O. Depending on implementation, physical I/O might be deferred until the smart pointer returned by D0Stream::read is dereferenced. The method D0Stream::test is a pure virtual function that is implemented in each concrete class to answer the question "Can I open the specified file?".

5.2 D0Stream factory classes

We envision a factory class D0StreamFactory for creating instances of D0Stream objects (Fig. 2). D0StreamFactory is a singleton class with a static instance method to return a pointer to itself. Concrete D0Stream subclasses would need to register themselves by calling the register method of D0StreamFactory. For each registered D0Stream subclass, D0StreamFactory will remember a type name and a pointer to that class's static test function. When users invoke the make_D0stream method of D0StreamFactory, they can specify a parcitular D0Stream type, or they can leave the type unspecified and let D0StreamFactory determine the correct type by calling the registered test functions. Make_D0Stream returns to the calling program a pointer to an instance of an already opened D0Stream object.

6 DØ Persistent Objects

As stated above, we intend that the DØ persistent object model conform to the ODMG persistent object model, except that ODMG classes, such as d_Objectare replaced by DØ defined classes such as d0_Object. This means persistent classes must be built according to the following rules.

1. Persistent classes inherit from the persistent base class d0_Object.
2. Persistent classes are composed of the following elements, and no others:
   (a) ODMG specified atomic types (d0_int, etc.).
   (b) Fixed length arrays.
   (c) Smart pointer and relationship types (d0_Ref<T>, etc.).
   (d) ODMG specified collection types (d0_Set<T>, etc.)
   (e) User-defined types composed of the above elements.

In this section we will look at how to implement the persistent object classes and related classes in a simple scheme with the following properties.

1. DSPACK is used for I/O.
2. DSPACK not used for memory management.
3. Events are read and written in their entirety.

We assume that there will be a base class D0Object for all DØ persistent object classes. D0Object inherits from d0_Object. All other persistent classes, such as D0Event, inherit from D0Object.

The sequence of events when event is read and written is as follows:

1. User calls D0Stream::read
   (a) DSPACK event is read from disk.
   (b) DSPACK event is converted into D0Event format.
2. User processes D0Event.
   (a) Create and/or modify persistent objects.
   (b) Mark selected objects for output using D0Stream::mark.
3. User calls D0Stream::write

   (a) D0Event is converted into DSPACK format.
   (b) DSPACK event is written to disk.

4. User deletes D0Event from memory.

The key steps in the above event processing loop are the conversion between D0Event and DSPACK format. This conversion will be handled by methods of concrete subclasses of D0Event that are invoked from the read and write methods of D0Stream. The user is not responsible to write these methods, but they will be generated automatically by a preprocessor from an object specification written in some object definition language. We assume that an initial version of the conversion will have a simple implementation which copies the data. Eventually, more sophisticated versions may attempt to convert the data in situ without copying.

6.1 Mapping the ODMG Persistent Object Model onto DSPACK

Each class is mapped onto one DSPACK data set. Multiple instances of the same persistent class are concatenated into a single data set. The name of the data set is the same as the name of the class (thus, persistent class names are limited to 16 characters).

The following rules are used to convert the data content of persistent and non-persistent classes into their DSPACK equivalents.

1. Atomic types are mapped into the corresponding DSPACK atomic types. Each atomic type has its own class (d0_intis a class, not a typedef).

2. Fixed length arrays are mapped into DSPACK fixed length arrays.

3. User defined types and mapped into corresponding DSPACK user defined types. Definitions of types can be nested.

4. Smart pointers (references and relationships) are replaced by DSPACK pointers.

5. Strings (d0_string) are broken out as a separate data set containing a single DSPACK dynamic string. A pointer and character count are stored in the parent data set.

6. ODMG collections are broken out as separate DSPACK data sets. A pointer and item count are stored in the parent data set.

The code to perform these conversions will be automatically generated by a preprocessor and appended to persistent class definitions.

As stated, the above rules are unambiguous when applied going from C++ classes to DSPACK data sets. However, some information is lost in the process. In the case of pointers and collections several different C++ constructs are be mapped into a single DSPACK construct. The reverse conversion depends on compiled in knowledge of

C++ classes to regenerate the missing information. The reverse conversion proceeds in top down fashion starting from all top level DSPACK data sets. Top level data sets are those that correspond to persistent classes. A special word "ds_istoplevel" is inserted to all top level data sets to distinguish them from non-top-level data sets.

References

[1] R. Zybert, DSPACK User's Guide, version 1.12.

[2] *The Object Database Standard: ODMG-93, Release 1.2*, R.G.G. Cattell ed. (Morgan Kaufmann, San Francisco, 1996).