AAF Object Manager Design Specification

Avid Technology

Revision 1.6

Author: Tim Bingham

June 4, 2001

Contents

1. Introduction

This document describes the design of the *Object Manager* component of the AAF software development kit.

2. Design Overview

The design treats an AAF file as a persistent object store. Persisted objects reside in a structured storage file. Once an AAF file has been opened, clients may access objects without having to be concerned about that file. Transient objects, not associated with any persistent store, may also be created. Transparent access to both persistent and transient objects is provided.

3. Summary of Requirements

The following sections summarize the currently known requirements placed on the *Object Manager*. These requirements are presented in no particular order.

3.1 Direct Support of the AAF Object Model

The *Object Manager* will directly support the AAF object model.

3.2 Persisting Objects and Object References

The *Object Manager* will provide isomorphic persistence of objects and their interrelationships. The persistence mechanism will properly handle circular and null references. The *Object Manager* will support persistence of the following kinds of object reference…

- Strong object reference – containment of one object by another object
- Strong object reference vector – containment of an ordered sequence of objects by another object
- Strong object reference set – containment of an unordered sequence of objects by another object
- Weak object reference - a reference to an object
- Weak object reference vector – an ordered sequence of references to objects
- Weak object reference set – an unordered sequence of references to objects

3.3 Transparent Access to Persistent Objects

The *Object Manager* will allow and require persistent objects to be accessed in exactly the same way as transient objects. In particular no explicit call will be needed to write an object to persistent store.

3.4 Implicit Saving of Individual Objects

The *Object Manager* will support implicit saving of modified objects to persistent store. This is a consequence of transparent access to persistent objects.

3.5 Explicit Saving of AAF Files

The *Object Manager* will support explicit saving of all modified objects associated with a given AAF file. Note that this requirement does not conflict with the requirement for transparent access to persistent objects since this requirement applies to a set of objects and not to a particular object.

3.6 Referential Integrity

In a valid AAF file all object references will be valid. A valid object reference is one that refers to an AAF object.

One form of an invalid object reference would be a dangling reference. A dangling reference is an object reference that was once valid but that has become invalid because the object to which it once referred has been deleted from the AAF file. Note that a null reference is not invalid. It is, however, an error to attempt to follow a null reference.

Because access to persistent objects is transparent this requirement also applies to an in-memory graph of AAF objects, persisted in an AAF file. Although in this case the requirement holds only at stable times. That is, the requirement holds between, and not during updates. The graph is stable before and after an update. A client application may only interrogate the graph of AAF objects at stable times and so can only see a stable graph.

3.7 Incremental File Access (Lazy Loading)

The *Object Manager* will support incremental access to an AAF file. To phrase this requirement in the negative - the *Object Manager* will *not* have to read an entire AAF file in order to access individual objects. Such incremental or lazy loading applies both at the object and at the property level. That is, object are not loaded into memory until they are referenced and even then the properties are not loaded until they in turn are referenced

3.8 File Integrity

The *Object Manager* will detect and robustly handle all structural errors in an AAF file that it opens for reading. The *Object Manager* will write only structurally valid AAF files.

3.9 Multiple Open AAF Files

The *Object Manager* will support an application having more than one AAF file open at the same time.

3.10 Lazy Loading

The *Object Manager* will support lazy loading of objects. An object will not be loaded until it is referenced, and, even then, objects referenced by the loaded object will not be loaded until they themselves are referenced. Lazy loading is optional and can be enabled or disabled dynamically on a per-AAF file basis.

3.11 Transient Objects

The *Object Manager* will support transient objects. A transient object exists independently of any AAF file. The state of a transient object is not saved to an AAF file. A given object is either persistent or transient. The *Object Manager* will define and enforce the rules for combining transient and persistent objects.

3.12 Object Extensibility

The *Object Manager* will support a hierarchy of classes, descended from a common ancestor AAFObject that is extensible in the following ways

- the definition of new classes
- the definition of new properties
- the definition of new property types
- the addition of optional properties to existing classes
- the definition of new classes with new behavior

The AAF class hierarchy is extensible but extensions are constrained to those described in the AAF Dictionary. Note that it is not possible to override the behavior of predefined classes.

3.13 Optional Properties

The *Object Manager* will support classes with optional properties. An optional property may or may not be present on individual instances of the class. The property definition must, however, be part of the definition of the class.

3.14 Edit in Place

The *Object Manager* will support edit in place. “Edit in place,” means the modification of a portion of an AAF file without having to read or rewrite the entire file.

3.15 Application Object Creation Model

The *Object Manager* will support an application object creation model with the following characteristics. This is referred to as "bottom up creation". That is, the application first creates an object and then later attaches that object to its containing object.

1. The *Object Manager* will allow client applications to specify the file with which a given persistent object is associated.
2. The *Object Manager* will ensure that objects from different files are not combined.
3. The *Object Manager* will allow client applications to specify that a given object be persistent or transient.
4. The *Object Manager* will ensure that persistent and transient objects are not combined inappropriately.

3.16 Forward Compatibility

Newer versions of the *Object Manager* must be able to read and reasonably process, files created by older versions.

3.17 Backward Compatibility

Newer versions of the *Object Manager* must write files that can still be processed reasonably by older versions.

3.18 Structural Checking

The *Object Manager* is responsible for checking and maintaining the structural integrity of an AAF file. The *Object Manager* is not responsible for checking and maintaining the correct semantics of an AAF file.

3.19 AAF File Byte Order

The *Object Manager* will support the AAF requirements for file byte order. These rules are

1) Byte Order is specified on a per file basis, that is, with the exception of certain media types (see below) all the data in a given file is in the same byte order
2) When a file is created it is created with the byte order of the host
3) When a file is modified the existing byte ordering of the file is preserved
4) Where a media format definition specifies the byte ordering for the media data the rules of the format definition are followed. Examples are
    a) AIFC
    b) WAVE

The *Object Manager* will perform byte reordering on read and/or write where the byte order of the file is different than that of the host. Such byte reordering is transparent to *Object Manager* client code. Note that these requirements apply to both predefined and user defined properties. Note also that these requirements apply only to the AAF property data itself and not to any meta-data needed by the *Object Manager* implementation.

3.20 AAF Files Contain Only AAF Objects

The *Object Manager* will enforce the constraint that AAF files contain only AAF objects. This means two things

1) The root IStorage in an AAF file is an AAF object. The *Object Manager* opens and creates structured storage files with `StgOpenStorage()` and `StgCreateDocFile()` rather than operating on an IStorage provided by clients. In other words the *Object Manager* is responsible for managing the entire structured storage file.
2) The *Object Manager* provides no interface that allows access to the IStorage representing a given AAF object. Therefore all IStorages in an AAF file are AAF objects.

3.21 AAF Files And Objects Are Not Embeddable

The *Object Manager* will enforce the constraint that AAF files and objects are not embeddable in other files. This constraint is realized by the fact that the *Object Manager* provides no interface that allows access to the IStorage representing a given AAF object.

3.22 Internal Interfaces To The *Object Manager*

The client of the *Object Manager* in the AAF reference implementation is the Data Model Manager. Since much of the Data Model Manager code will have originated in the OMFI implementation, the interfaces presented to client code by the *Object Manager* will make the porting process as simple as possible.

3.23 External Interfaces To The *Object Manager*

The *only* client of the *Object Manager* in the AAF reference implementation is the Data Model Manager. There is no requirement for the *Object Manager* to export interfaces that are visible to clients of the AAF reference implementation. This means that even though clients of the AAF reference implementation will use COM to call the AAF API, there is no requirement that the *Object Manager* implementation use COM.

3.24 Admissibility Of Alternative Implementations Of AAF

The *Object Manager* design will allow for alternative implementations of AAF. In particular

- The same AAF object created by different implementations of AAF should have the same external representation.
- Once created, an AAF file must not be tied to a particular AAF implementation.
- The objects in an AAF file must not be identified by COM class ids (code class ids) which are specific to a particular AAF implementation, instead they must be identified by SMPTE stored object ids which are the same for all AAF implementations.

3.25 File Size

The *Object Manager* will create AAF Files that are approximately the same size as an equivalent OMF file. This may be difficult to accomplish given a Structured Storage implementation with "blocky" allocation.

3.26 Extensible Types

The *Object Manager* will support the run-time extension of the set of possible property types. That is, the set of possible property types must not be compiled into the *Object Manager.*

3.27 Canonical Types

Some types, notably `enum` and `struct` types from `C` and `C++` have in memory representations that differ from compiler to compiler. For `enum` types different sizes may be chosen. For `struct` types different alignment constraints may be applied. These differences in representation most often occur between different platforms but sometimes arise between different compilers on the same platform and even within the same compiler on a given platform as the result of applying different compile time options. The *Object Manger* must support persisting values of these types in a platform and compiler independent way.

3.28 Performance and Scalability

The Object Manger will meet the following performance goals…

- file open latency independent of the number of objects in the file
- file close latency independent of the number of objects in the file (note that close does not imply save)
- file save time linear with the number of modified objects
- access to set elements logarithmic with the number of objects in the set

3.29 Media Data

The *Object Manager* must support the storage and retrieval of media data in AAF files. The usual read, write and seek operations must be supported.

3.30 Portability

The *Object Manager* portability requirements are the same as for the entire reference implementation. Those requirements are summarized here. The *Object Manager* code must be portable to the following platforms…

- Windows NT – x86
- Macintosh – PPC
- Silicon Graphics Irix – MIPS

In addition the *Object Manager* should not be an obstacle to adopters wishing to port the reference implementation to other platforms. These requirements boil down to…

- Use of ANSI conformant C++
- Limiting dependencies on no-AAF technologies to the dependency on Microsoft Structured Storage.

3.31 Client Specified Unique Identifiers

Objects in a reference set need to be uniquely identified. The *Object Manager* will allow the client code (the Data Model Manager, or DM, code) to specify the unique identifier, usually a GUID, to be used for a particular object. Examples of such unique identifiers are…

- a MOB id

- a SMPTE assigned unique label

3.32 Other Requirements

[TBS. As other requirements are identified they will be summarized here.]

4. Overview of Structured Storage

A structured storage file may be compared to a file system with in a file. The structured storage concepts map to file system concepts as follows

- IStreams are approximately equivalent to file system files. In particular an IStream may be used to store arbitrary data just as a file may be used to store arbitrary data.
- IStorages are approximately equivalent to file system directories. In particular an IStorage may contain both IStreams and other IStorages just as a directory may contain both files and other directories.

5. Design Principles

- Ensure the consistency of the stored format of AAF objects, predefined and extended, by providing automatic persistence of registered properties. This approach neither requires nor allows individual objects or classes to manage their own stored representations.
- AAF Files are not a general-purpose object store. Only objects that are described in the AAF Dictionary may be stored in and retrieved from an AAF file.
- Support natural coding of AAF class methods when accessing persistent properties. That is, no explicit calls are needed to access properties. Accessing a persistent variable should be as similar as possible to accessing a non-persistent variable of the same type. This principle is intended to support the migration of the OMFI tool-kit code into the AAF tool-kit by allowing natural coding of persistent property access.
- Independent of the particular AAF object model or class hierarchy chosen.
- No per-class code to support persistence.
- Construct only legal object networks and files rather than “construct then verify”.
- Loading an object reference is the same as performing a lazy load of the referenced object. Dereferencing an object reference causes the actual load to occur.
- [TBS. As other design principles are identified they will be recorded here.]

6. Class Interfaces

This section describes the interfaces to the principal classes in the *Object Manager.*

6.1 Defining and Accessing Properties

This section describes how properties are defined and accessed by the toolkit implementation code. The example uses a sub-set of the properties of the AAFHeader class.

6.1.1 Framework

To become persistent a class must inherit from class OMStorable. Class OMStorable contains the declaration of the persistent property set and member functions for saving and restoring the properties.

```
class OMStorable {

…
protected:
```

```
  void saveTo(OMStoredObject& s) const;
  void restoreContentsFrom(OMStoredObject& s);

  OMPropertySet _persistentProperties;

};

class ImplAAFObject : public OMStorable, public ImplAAFRoot
{
…
};
```

6.1.2 Property Declaration

A class must declare its properties using template classes exported by the *Object Manager*. This gives the properties persistence behavior but allows the implementation code to access them naturally.

```
class ImplAAFHeader : public ImplAAFObject
{
…
private:
  // Persistent properties
  //
  OMFixedSizeProperty<aafInt16>                            _byteOrder;
  OMFixedSizeProperty<aafTimeStamp_t>                      _lastModified;
  OMStrongReferenceVectorProperty<ImplAAFIdentification> _identificationList;

  // Non-persistent properties
  //
  … declared as for regular C++ …

};
```

6.1.3 Property Access

```
AAFRESULT STDMETHODCALLTYPE
    ImplAAFHeader::GetByteOrder (aafInt16 *pByteOrder)
{
   *pByteOrder = _byteOrder;
   return AAF_ERR_NONE;
}
```

6.1.4 Property and Property Set Initialization

The constructors for all properties take a property id (PID_ value) and a property name. The constructor for a persistent object must…

1. initialize the properties it contains, and
2. initialize its own property set by inserting the properties

Here is some example code from AAFHeader that illustrates this…

```
ImplAAFHeader::ImplAAFHeader ()
: // Initialize the properties of this object
  //
  _byteOrder(         PID_Header_ByteOrder,          "ByteOrder"),
  _lastModified(      PID_Header_LastModified,       "LastModified"),
```

```
  _identificationList(PID_Header_IdentificationList, "IdentificationList"),
{
  // Initialize the property set of this object by
  // inserting the properties of this object.
  //
  _persistentProperties.put(_byteOrder.address());
  _persistentProperties.put(_lastModified.address());
  _persistentProperties.put(_identificationList.address());
  …
}
```

Note that `OMPropertySet::put()` accepts the address of an `OMProperty` object. The code above calls `OMProperty::address()` to obtain the correct address since `OMProperty::operator &` is overridden to provide the address of the property *data* (rather than the property *object*) for the convenience of AAF class member functions.

6.2 Saving and Restoring Property Values

Clients of the *Object Manager* do not need to provide any code that explicity saves or restores individual property values. The saving and restoring of property values is a file level operation. When a file is opened, using `OMFile::open()`, the property values are lazily restored, when a file is saved, using `OMFile::save()`, any dirty (changed) properties are written to the file. Changed property values may also be written to the file by an explicit call to `OMFile::save()`.

6.3 Persistent Property Class Hierarchy

The hierarchy of property classes is as follows:

```
OMProperty
  OMReferenceProperty<ReferencedObject>
    OMWeakReferenceProperty<ReferencedObject>
    OMStrongReferenceProperty<ReferencedObject>
  OMSimpleProperty
    OMVectorProperty<PropertyType>                      // Not yet implemented
    OMFixedSizeProperty<PropertyType>
    OMVariableSizeProperty<PropertyType>
      OMCharacterStringProperty<CharacterType>
        OMStringProperty
        OMWideStringProperty
  OMCollectionProperty
    OMStrongReferenceVectorProperty<ReferencedObject>
    OMWeakReferenceVectorProperty<ReferencedObject>    // Not yet implemented
    OMStrongReferenceSetProperty<ReferencedObject>     // Not yet implemented
    OMWeakReferenceSerProperty<ReferencedObject>       // Not yet implemented
  OMDataStreamProperty
```

6.4 Creating Object Instances

When reading an AAF file the *Object Manager* needs to be able to create object instances based on the stored object id found in the file. The following *Object Manager* interface is defined for this purpose:

```
class OMClassFactory {
public:

  // Create an instance of the appropriate derived class, given the class id.
  //
  virtual OMStorable* create(const OMClassId& classId) const = 0;
```

```
};
```

The function OMClassFactory::create() creates an uninitialized instance of the class with the given *classId.* This interface is defined by the *Object Manager*, called by the *Object Manager* and implemented by the *Object Manager* client. The *Object Manager* reads a stored object id from the AAF file and passes it to `OMClassFactory::create()`, this function returns a pointer to a new instance of the appropriate class with a properly initialized property set (including and extension properties). The phrase “appropriate class” is intended to include any extension classes and any "default base class instantiation" that is necessary. The phrase “properly initialized property set” is intented to include any extension properties that may be present.

The *Object Manager* client is also responsible for associating a given class factory with a given file using the open/create methods on `OMFile`.

```
class OMFile : public … {
public:
…
  static OMFile* openExistingRead(const wchar_t* fileName,
                                  const OMClassFactory* factory,
                                  const OMLoadMode loadMode);
…
};
```

The client of the *Object Manager* in the AAF reference implementation is the Data Model Manager. The Data Model Manager implements OMClassFactory::create() with reference to the appropriate AAFDictionary.

6.4.1 Creating Objects and Meta Data Objects

This section outlines some design rules to be observed when creating objects, and the associated meta-data objects.

1) No `OMProperty` may be created withouth having a valid `OMPropertyDefinition*`
2) No property value may be read or written without having a valid OMType `*`, this is obtained from the `OMPropertyDefinition*` supplied when the `OMProperty` was created.
3) All properties to be created as required properties (as opposed to optional properties) prior to property initialization.
4) After an object is created (instantiated), but before its properties are initialized the meta-data for that object must exist in memory in a valid state. This meta-data includes
   a) a ClassDefinition for the object and all of its ancestor classes
   b) a PropertyDefinition for each property in each of the above ClassDefinitions
   c) a TypeDefinition for each of the above PropertyDefinitions

The meta-data is created on demand. The meta-data for Class Foo is created the first time an instance of class Foo is created, subsequent instances of class Foo share the same meta data as all previous instances.

5) There are two kinds of built in information.
   a) The information needed to create "standard" AAF objects that are not defined in the current file (in the case of `CreateNewModify()` this means all classes).
   b) The information that cannot be read from the file because it is either intrinsic (e.g. an object is an IStorage) or self describing (e.g. the instance of ClassDefinition that describes a ClassDefinition object or the Name property of TypeDefinitionString which is itself of a type described by TypeDefinitionString).
6) This results in the following Axioms
   a) Create function
      i) File (should have no SOID)
      ii) Header
      iii) Dictionary
   b) LookUpClass function
      i) ClassDefinition
   c) LookUpType function
      i) StrongReference

ii) StrongReferenceVector
iii) StrongReferenceSet
iv) WeakReference (AUID)
v) WeakReferenceVector (AUID array)
vi) WeakReferenceSet (AUID array)
vii) AUID
viii) String

d) LookUpProperty function
i) ?

7) The axiomatic definitions are
a) Never read from the file.
b) Written to the file for consistency.

Doing this requires an api other than LookUp. An api is needed that will determine if an object is present in the file without causing the object to be loaded if it is in fact present. e.g. `bool OMStrongReference{Set|Vector}<>::isPresent(AUID ident);`

8) The axiomatic definitions are implemented as follows. This example uses LookUpType, however, there are other axiomatic definitions.

```
LookUpType(a)
{
  if (a == axiomatic type 1) {
    create type 1 in memory
    ensure type 1 is present in file dictionary
    result = type 1
  } elseif (a == axiomatic type 2) {
    create type 2 in memory
    ensure type 2 is present in file dictionary
    result = type 2
  } elseif (a == ...) {
   ...
  } elseif (a == axiomatic type N) {
    create type N in memory
    ensure type N is present in file dictionary
    result = type N
  } else {
    if (a is registered in dictionary) {
      result = type of a // type is unpersisted here
    } else {
      create type of a
      register type of a
      result = type of a
    }
  }
}
```

9) As a debugging/diagnostic aid the implemetation should detect loops caused by an incorrect implementation (e.g. missing axiom). One way to do this is as follows.

```
  Stack t;

  CreateDefinition(def, ident)
  {
    if (t.contains(ident)) {
      error
    } else {
      t.push(ident)
```

```
            ... creation code ...
            o = create def
            o.SetIdentification(i)
            ...
          t.pop(ident)
    }
```

10) An alternative scheme would be to use 3-stage object creation (instantiate, register and initialize) as a means of breaking cycles. That approach has the disadvantage that an object may be accessed after being instantaited and registered but before being initialized. Also it does not prevent by itself, attempting to read axiomatic definitions from the file.

6.5 Type-specific Byte Reordering, Internalization and Externalization

There are at least two approaches to choosing the persisted representation of structured types. Here "structured types" includes structs, arrays and enums - anything where a compiler has a choice of the in-memory representation. The compiler choices are as follows…

a) The size of a data item
b) The alignment of a data item (and consequently padding)

Note also the repertoire of types supported by the *Object Manager* is not compiled-in, but is instead available at run-time as described in the dictionary.

The two possible approaches are

a) persist using the (possibly byte-swapped) in memory representation of the host platform (machine + OS + compiler) and describe that representation in the dictionary.
b) use an AAF defined canonical representation.

The approach chosen by the *Object Manager* is b) since it is

a) simple to explain and document
b) in accordance with the plan for SMPTE standardization
c) platform (machine + OS + compiler) neutral and so fosters interchange

6.5.1 Definition of `OMType`.

These requirements result in an interface used by the *Object Manager* for manipulating data types at run time. `Class OMType` is defined as follows.

```
class OMType {
public:

  virtual void reorder(...);
  virtual void externalize(...);
  virtual void internalize(...);

};
```

The purpose of each virtual function is as follows …

a) `reorder() -` put the bytes of the data into the proper order, "byte swap". By convention the data is taken to be in external form.
b) `externalize() -` put the bytes of the data into the appropriate external from, typically this involves removing any padding (for record types), and adjusting the size of the data (for enumerated types).
c) `internalize() -` put the bytes of the data into the appropriate internal from, typically this involves adding any required padding (for record types), and adjusting the size of the data (for enumerated types).

Note this design uses `externalize()` and `internalize()` instead of, say, `pack()` and `unpack()` since `externalize()` may in fact make a data value larger. For example, `aafBool` compiles to 1 byte with Macintosh CodeWarrior and to 2 bytes with NT VC++ (with our current VC++ and CodeWarrior compiler settings) and so if we choose to make all AAF, non-extensible, enumerations 2 bytes in size, then `externalize()` on the Macintosh will cause the value to be *increased* in size. So clearly `externalize()` is to be preferred over `pack()`.

6.5.2 Reading and Writing Values Described by `OMType`

Given the declarations

```
OMType* type;
OMByte* value;
```

Persisting a data value is

```
type->externalize(value);
if (fileByteOrder != hostByteOrder)
  type->reorder(value);
write(value);
```

Unpersisting a data value is

```
read(value);
if (fileByteOrder != hostByteOrder)
  type->reorder(value);
type->internalize(value);
```

All type definitions are implemented in the *Data Manager* and descended from `OMType` as follows …

```
class ImplAAFTypeDef : public OMType {
 ...
};
```

Note that reordering is always performed on values in their external form.

7. Property Types

These types are described in detail in Annex B of the document “Proposed SMPTE Recommended Practice for Television - Interchange of Video and Audio Material and Related Descriptive Information as Edit Decision Data”. The classification of the types presented here is introduced by this document.

7.1 Structural Types

The structural types define the structure of a given AAF object network. These types are built in to the *Object Manager*.

Type	Meaning
Strong reference	Containment (ownership) of an object
Strong reference vector	Containment (ownership) of an ordered collection of objects
Strong reference set	Containment (ownership) of an unordered collection of objects
Weak reference	Pointer to an object
Weak reference vector	Ordered collection of pointers to objects

Weak reference set	Unordered collection of pointers to objects

7.2 Primitive Types

The primitive types are the fundamental types supported by the *Object Manager*, all other types are defined in terms of these. These types are built in to the *Object Manager.*

7.3 Compositional Types

Used to build define a type in terms of other, previously defined types. These types are built in to the *Object Manager.*

7.4 Composed Types

7.4.1 Types Not Specific To AAF

These types are not specfic to AAF. These types are not known to the *Object Manager.* These types are built in to the AAF reference implementation. They are defined in terms of the built-in types.

7.4.2 Types Specific To AAF

These types are specfic to AAF. These types are not known to the *Object Manager.* These types are built in to the AAF reference implementation. They are defined in terms of the built-in types.

7.5 Summary Of Property Types

1) Primitive Types
    a) Integer (parameterised by size and signedness)
2) Compositional Types
    a) VaryingArray
    b) FixedArray
    c) Aggregate
    d) Renamed
    e) Enumerated
    f) String
    g) Stream (DataValue, essence or media data)
3) Composed types
    a) Types Not Specific To AAF
        i) Character types
            (1) Character (UInt16 - Unicode character)
        ii) Enumerated types
            (1) Boolean
        iii) Signed numeric types
            (1) Int8
            (2) Int16
            (3) Int32
            (4) Int64
        iv) Unsigned numeric types
            (1) UInt8
            (2) UInt16
            (3) UInt32
            (4) UInt64
        v) Signed numeric array types
            (1) Int8Array
            (2) Int16Array
            (3) Int32Array
            (4) Int64Array
        vi) Unsigned numeric array types
            (1) UInt8Array

(2) UInt16Array
(3) UInt32Array
(4) UInt64Array

b) Types Specific To AAF
   i) Enumerated types
      (1) ColorSitingType
      (2) EdgeType
      (3) EditHintType
      (4) FadeType
      (5) FilmType
      (6) JPEGTableIDType
      (7) LayoutType
      (8) ProductVersion
      (9) PulldownKindType
      (10) PulldownDirectionType
      (11) PhaseFrameType
      (12) TapeCaseType
      (13) TapeFormatType
      (14) VideoSignalType
      (15) TCSource
      (16) ReferenceType
   ii) Aggregate types
      (1) TimeStamp
      (2) VersionType
      (3) Rational
      (4) Rectangle
   iii) Renamed types
      (1) Length
      (2) Position
   iv) Array types
      (1) CompCodeArray
      (2) CompSizeArray

7.6 How Types Are Composed

Composed types (instances of compositional types) contain pointers to the types of which they are composed. For example, a fixed array of integers is prepesented by an instance of an array that contains a count of the number of elements and a pointer to an integer type.

The compositional types support the same polymorphic interface as the primitive types. This is an example of the "Composite" design pattern.

As an example consider a `reorder()` method to reorder ("byte swap") a value of that type. For a primitive type, such as an integer, the `reorder()` method is implemeted as a direct manipulation of the bytes of the value. For a composed type, say a fixed array of integer, the `reorder()` method is implemented by multiple `reorder()` calls on the composed type, in this case integer. In this example the array `reorder()` method is simply a loop that calls the `reorder()` method for the array element type.

7.7 Mapping Of Types To Structured Storage

This table shows how the property types map to the different stored forms supported by the *Object Manager* when mapping property values to structured storage.

Type	Mapped to Structured Storage by *Object Manager* as
Strong reference	SF_STRONG_OBJECT_REFERENCE
Strong reference vector	SF_STRONG_OBJECT_REFERENCE_VECTOR
Strong reference set	SF_STRONG_OBJECT_REFERENCE_SET

Weak reference	SF_WEAK_OBJECT_REFERENCE
Weak reference vector	SF_WEAK_OBJECT_REFERENCE_VECTOR
Weak reference set	SF_WEAK_OBJECT_REFERENCE_SET
Media data	SF_DATA_STREAM
All other data types	SF_DATA

See section 8.4 of this sepcification for more details on the mapping of AAF property types on to structured storage. The SF_* values are passed as the "`storedForm`" parameter in the `OMStoredObject::read()` and `OMStoredObject::write()` calls.

Note that the SF_STRONG_OBJECT_REFERENCE_SET and SF_WEAK_OBJECT_REFERENCE_SET stored forms are not yet implemented. The VECTOR form is used in all cases. The only consequence of this is that ordering is preserved where it need not be.

7.8 Indirect, private, encrypted, opaque and KLV types

This section describes a design solution for the representation of the following data types in AAF.

1) The following three uses of DataValue
   a) "stream" - as in EssenceData::Data and TimeCodeStream::Source
   b) "varying array of bytes" - as in AIFC::Summary, EdgeCode::Header, RGBADescriptor::Palette, TIFFDescriptor::Summary and WaveDescriptor::Summary
   c) "void *" - as in ConstantValue::Value, ControlPoint::Value and TaggedValue::Value
2) "private types" - that is data whose type is known to party A, the creator of the data, and to party B the consumer of the data, but not to party C who may modify the file containing the data being communicated from A to B. The data is private to A and B. The private data must be preserved, without client code intervention, when C modifies the file in which it resides.
3) "encrypted types" - The same as "private data" except that the data is encrypted. While A and B trust each other, neither trusts C. Again the data must be preserved across modifications made by C.
4) "SMPTE KLV types"

7.8.1 Design proposal

7.8.1.1 DataValue

7.8.1.1.1 DataValues Representing a “stream"

Such properties should not be implemented as DataValues. This is already implemented, streams are described by AAFTypeDefStream and implemented by OMDataStreamProperty.

7.8.1.1.2 Data Values Representing "array of bytes"

Such properties should not be implemented as DataValues. Arrays of bytes are described by AAFTypeDefVariableArray (or by AAFTypeDefFixedArray) with an element type of "unsigned 8-bit integer" and implemented by OMVariableSizeProperty<OMByte>

7.8.1.1.3 DataValues representing "void *"

These values are represented by AAFTypeDefIndirect. Values described by AAFTypeDefIndirect consist of

1) the AUID of the actual type
2) the value as a variably sized array of bytes

The class AAFTypeDefIndirect has no properties of its own other than a reference to the actual type.
Properties values described by AAFTypeDefIndirect are automatically byte swapped by the Object Manager just like any other type.
The class AAFTypeDefIndirect has a method, ActualType() which returns the actual type of a given value. This works by retrieving the AUID from the value and looking it up in the dictionary. It is required that the AUID represent a type in the dictionary otherwise the file is invalid.

This design satisfies the requirement that the type of a property does not change, since properties of this type are always described as AAFTypeDefIndirect.
This design allows the actual type of a property to vary on a property instance by property instance basis by storing the AUID for the actual type along with the property value.

7.8.1.2 "private types"

These values are represented by AAFTypeDefPrivate. The design for AAFTypeDefPrivate is identical to that of AAFTypeDefIndirect except for the following

1) Values described by AAFTypeDefPrivate also contain the byte order in which the property value was first created.
2) Properties values described by AAFTypeDefPrivate are only byte swapped by the Object Manager when accessed by a party who knows the actual type.
3) The ActualType() method can fail when applied to a legal file. This happens if party C (to whom the type is private) tries to access the property.
4) The ActualType() method succeeds if the type definition corresponding to the AUID in the property value is first installed in the dictionary, for example by party B (to whom the type is not private).

7.8.1.3 "encrypted types"

These values are described by AAFTypeDefEncrypted. The design for AAFTypeDefEncrypted is identical to that of AAFTypeDefPrivate except for the following

1) Values described by AAFTypeDefEncrypted also contain a GUID, for use by trusted parties to identify the encryption method (or encryption key).
2) After calling ActualType() trusted clients must first decrypt the data before using the actual type to interpret it..

7.8.1.4 "SMPTE KLV types"

These values are described by AAFTypeDefSMPTEKLV. Values described by AAFTypeDefSMPTEKLV consist of

1) the byte order in which the property value was first created
2) the SMPTE value V as a variably sized array of bytes

The class AAFTypeDefSMPTEKLV has the following properties

1) the SMPTE key K as an unsigned integer
2) an AUID identifying the type of the value if such a type has been defined or, if not, identifying AAFTypeDefOpaque (see below)

Properties values described by AAFTypeDefSMPTEKLV are automatically byte swapped by the Object Manager just like any other type (unless their type is described by AAFTypeDefOpaque).
The class AAFTypeDefSMPTEKLV has a method, ActualType() which returns the actual type of a given value. This works by looking up the AUID (a property of AAFTypeDefSMPTEKLV) in the dictionary.
Thus instances of AAFTypeDefSMPTEKLV define a mapping from SMPTE keys onto AAF types that describe the associated SMPTE values. This is a many to one mapping. That is, several different SMPTE keys may map to the same AAF type.
Initially values can be described by AAFTypeDefOpaque, these can then later be described by appropriate AAF predefined types without invalidating any previously stored values. In this way AAF files may contain KLVs with unknown keys (This is the whole purpose of KLV).
Note that the SMPTE length L is not stored explicitly, instead it is an attribute of the value.

7.8.1.5 AAFTypeDefOpaque

This type allows values whose type is unknown (temporarily or permanently) safely to be stored in AAF files. The value of a property described by AAFTypeDefOpaque consists of

1) the byte order in which the property value was first created
2) the value as a variably sized array of bytes.

7.9 Name Equivalence

Unfortunately the names for various *Object Manager* concepts have evolved differently in different documents. The following table shows the equivalence between the various names.

Concept	Object Spec	Original SMPTE draft	Latest SMPTE draft	Symbolic Name
Strong Reference	ObjRef	ObjRef	StrongRef	SF_STRONG_OBJECT_REFERENCE
General Weak Reference	None	None	None	None

Restricted Weak Reference	ObjRef	RefAUID	WeakRef	SF_WEAK_OBJECT_REFERENCE
Strong Reference Vector	ObjRef	ObjRefArray	StrongRefArray	SF_STRONG_OBJECT_REFERENCE_VECTOR
General Weak Reference Vector	None	None	None	None
Restricted Weak Reference Vector	ObjRef	RefAUIDArray	WeakRefArray	SF_WEAK_OBJECT_REFERENCE_VECTOR
Strong Reference Set	None	None	StrongRefSet	SF_STRONG_OBJECT_REFERENCE_SET
General Weak Reference Set	None	None	None	None
Restricted Weak Reference Set	None	None	WeakRefSet	SF_WEAK_OBJECT_REFERENCE_SET

8. *Object Manager* Design

The following sections describe the design of the principle aspects of the *Object Manager*.

8.1 *Object Manager* Interfaces

This section describes the interfaces to the *Object Manager*.

Access to most *Object Manager* functionality is provided implicitly from the perspective of tool kit clients. Exceptions to this are the AAFSession, AAFile and AAFClassDictionary classes which allow tool kit clients explicit access to *Object Manager* functionality.

8.1.1 Interfaces to Support the Tool Kit Implementation

The *Object Manager* will provide the following explicit interfaces for use by the implementation of the tool kit.

8.1.1.1 Definition Classes

The *Object Manager* must support both predefined and user defined AAF classes. This design proposes that instances of AAF definition classes are used to represent both predefined and user defined AAF classes.

8.1.1.2 Equivalence of Predefined and User Defined AAF Classes

The AAFClassDictionary class manages instances of these definition classes. There is one instance of the AAFClassDictionary class for each AAF file.

An application must register any extensions it makes to AAF with the tool kit. This design proposes that the same underlying mechanism is used for registering both predefined and user defined AAF classes. This design also proposes that the information for the definition of the predefined classes be derived from the same source as the class declarations for the AAF classes. A set of preprocessor macros will be provided for use in declaring and defining the predefined AAF classes. This set of macros will, in effect, call the same API functions that tool kit clients will call when creating extension AAF classes. However, in the case of the predefined AAF classes, certain optimizations will be provided. These optimizations include

1) No consistency checking at application run time, except in the debug version of the tool kit.
2) Registration of predefined AAF classes takes place at compile time instead of at run time. This will be hidden by the macros.
3) Instances of the definition objects that describe the predefined AAF classes will be shared across the class dictionaries associated with different AAF files. If an application chooses to modify, by extension, an existing definition, for example, by adding a property to one of the predefined AAF classes, then the definition object will be copied prior to extension. This is an implementation of “copy on write” semantics.
4) Instances of the predefined definition classes are not persisted; this is the very definition of “predefined”. An exception is the case of an extended predefined class.

The provided macros will include the following. See the source file `AAFMetaDictionary.h` for more details.

1) `AAF_TABLE_BEGIN()` - Begin a table of AAF class and property definitions.

2) `AAF_TABLE_END()` - End a table of AAF class and property definitions.

3) `AAF_CLASS(name, id, parent)` - Define an AAF class.

```
name      = the name of the class
id        = the auid used to identify the class
```

```
parent    = the immediate ancestor class
```

4) `AAF_CLASS_END(name)` - End an AAF class definition.

```
name      = the name of the class
```

5) `AAF_CLASS_SEPARATOR()` - Separate one AAF class definition from another.

6) `AAF_PROPERTY(name, id, tag, type, mandatory, container)` - Define an AAF property.

```
name      = the name of the property
id        = the auid used to identify the property
tag       = the short form of the id
type      = the type of the property
mandatory = true if the property is mandatory
            false if the property is optional
container = the class that defines this property
```

8.1.1.2.1 Class Definition

[TBS. This section will describe the API provided by the *Object Manager* for use in defining classes.]

8.1.1.2.2 Property Definition

[TBS. This section will describe the API provided by the *Object Manager* for use in defining properties.]
There will be no separate API for creating property definition objects. Property definition objects cannot be created without reference to the class to which they belong.

8.1.1.3 Property Access

[TBS. This section will describe the API provided by the *Object Manager* for use in accessing properties.]

8.1.1.4 Media Stream Access

Clients of the tool kit will be able to access both external and embedded media data via the AAFMediaStream class. Once an instance of the AAFMediaStream class has been created, internal and external media are accessed transparently.

8.1.1.5 Media Stream Access Functions

[TBS. This section will describe the *Object Manager* interfaces that support the AAFMediaStream class.]
The *Object Manager* will provide the following operations on media streams.

1. Open
2. Close
3. Read
4. Write
5. Seek
6. GetPosition
7. SetPosition

8.1.2 Interfaces Used By The *Object Manager*

The *Object Manager* will use the interfaces described in the following sections.

8.1.2.1 Structured Storage

The *Object Manager* will use the IStorage and IStream interfaces and the StgOpenStorage and StgCreateDocfile APIs. Only functionality supported by the reference implementation of Structured Storage will be used.

8.2 File Level Operations

The *Object Manager* provides the following file level operations.

1) Open an existing file for reading only.
2) Open an existing file for modification.
3) Create an empty file.
4) Create a transient file.
5) Close an open file. On close no save is implied, changes to a file must be saved explicitly.
6) Save changes to an open file. That is, write out all dirty objects.
7) Control lazy loading. These modes must be specified when the file is opened. There are two levels of lazy loading.
    a) None (or eager loading) – the entire contents of the file is loaded when it is opened. That is, all strong references are followed. This mode is useful, for example, in applications that cannot tolerate loading delays while processing a group of objects. In this mode memory reclamation or "lazy unloading" is not performed.
    b) Object granularity (or lazy loading) – when an object is loaded, the whole object is loaded. This means loading all properties except media. References are not followed. This mode is useful, for example, when accessing all of the properties of a few of the objects in an AAF file. In this mode memory reclamation or "lazy unloading" is performed.
8) Revert. Discard any changes made to the file since the last open or save operation.

8.2.1 Semantics of AAFFile::Save() and AAFFile::Close()

This section is a summary of the semantics of AAFFile::Save() and AAFFile::Close() as they relate to unsaved changes and to read-only files. In this section *client* means AAF API client and not *Object Manager* client.

1) Clients should call AAFRoot::ReleaseReference() on the head object before calling AAFFile::Close().
2) AAFFile::Close() does not call AAFFile::Save().
3) To ensure that objects are saved clients must explicitly call AAFFile::Save() before calling AAFFile::Close(). This is because of point 2.
4) AAFile::Close() calls AAFRoot::ReleaseReference() on the head object.
5) AAFFile::Save() silently ignores unsaved objects (The objects are not saved and the HRESULT is success).
6) AAFFile::Save() on a read-only file always fails.
7) AAFFile::Save() on a transient file always fails.
8) AAFFile::Save() silently ignores unsaved objects associated with a read-only file. This is a consequence of point 5 which applies equally to read-only and to writeable files.
9) An application may open a read-only file and modify the objects associated with that file. The modifications cannot be saved to the original file (Because of point 6). The modifications must either be written to another file with AAFFile::SaveAs() or moved/copied to another writeable file and then saved.
10) An application may discard all unsaved changes by calling AAFFile::Revert(), provided that the file was opened in revertable mode.
11) Media data (essence) is treated as a special case.
    a) Changes to media data are never revertable
    b) Changes to media data take place immediately in advance of any call to AAFFile::Save()
    c) Any attempt to write media data to or change media data in a read-only file fails immediately.

8.2.2 File Mode Flags

The following bits are defined in the modeFlags argument to the following calls
- AAFFileOpenExistingRead ()
- AAFFileOpenExistingModify()
- AAFFileOpenNewModify ()

Public bit definitions
- kAAFFileModeUnbuffered - to indicate buffered mode. Default is buffered.
- kAAFFileModeRevertable - to indicate that Revert is possible on this file (for all changes except those to essence).

- kAAFFileModeEagerLoad - to indicate that the objects in the file should all be loaded when the file is opened . The default is lazy loading in which the objects are loaded on demand.
- kAAFFileModeReclaimCleanObjects - to indicate that the memory associated with clean in-memory objects may be relaimed. The default is never to reclaim the memory associated with clean objects.

Private bit definitions (to help with performance optimizations)
- kAAFFileWriteProperties - to indicate that objects be written out one property at a time. The default is to write whole objects. (undocumented)
- kAAFFileModeKeepObjectsOpen - to indicate that the IStorage for all objects should be kept open. The default is to keep the IStorage for all objects closed, except during Save(). (undocumented).
- kAAFFileModeWriteAllObjects - to indicate that the dirty bit be ignored on Save(). The default is to write only dirty objects on Save(). (undocumented)
- kAAFFileSeparateIndexAndValue - to indicate that the property set index and the property values of an object be stored in separate IStreams. The default is to combine them into a single IStream. (undocumented)
- kAAFFileModeUseLargeSectors - to indicate that the 4k (large) sector size implementation of structured storage be used (if available). The default is to use the small sector size implementation. Warning - use of this flag creates files that are incompatible with the small sector size implementation of structured storage (undocumented).
- KAAFFileCloseFailDirty – to indicate that Close() should fail if there are dirty objects. (undocumented)

8.3 Persistence Infrastructure

8.3.1 Persistence Rules by Type

The actions required to persist data depend upon the type of that data. The *Object Manager* needs to able to persist the following classes of data

1. Property values. There are two kinds of property value
   - 1.1. Ordinary property data, both simple and structured
   - 1.2. Media data
2. Object references. There are six kinds of object reference
   - 2.1. Strong object references
   - 2.2. Strong object reference vectors
   - 2.3. Strong object reference sets
   - 2.4. Weak object references
   - 2.5. Weak object reference vectors
   - 2.6. Weak object reference sets

8.3.1.1 Property Values

The primitive, or built-in, types form the base vocabulary of types. All properties are defined in terms of the primitive types.

8.3.1.1.1 Ordinary Property Data

Ordinary property data is stored in an IStream contained in the IStorage that represents the object that contains the property data. The property index, in a separate index IStream, describes the offset and extent of the property data in the property value IStream.

8.3.1.1.2 Media Data

Media data is stored in an IStream contained in the IStorage that represents the object that contains the media data.

8.3.1.2 Object References

Object references are the mechanism for describing associations between objects. All associations between objects are described using object references.

8.3.1.2.1 Strong Object References

A strong reference denotes containment and ownership of one object by another object.

To persist a strong object reference, the strong object reference is followed. If the referenced object is has not already been persisted, it is persisted by applying these rules recursively. This is recursive persistence. In this design an optimization to recursive persistence is proposed. All dirty objects can be found without traversing the entire tree of strong references. The object directory contains one entry for each object that is in memory. Additionally the dirty objects are tagged as such. Consulting the object directory allows all the dirty objects to be found without traversing the entire tree of strong references.

8.3.1.2.2 Strong Object Reference Vectors

A strong object reference vector denotes containment and ownership of a collection of objects by another object.

A strong object vector is persisted by iterating over the elements of the vector and treating each as a strong object reference.

8.3.1.2.3 Strong Object Reference Sets

8.3.1.2.4 Weak Object References

A weak reference denotes a general, possibly shared, association from one object to another.

To persist a weak object reference an external representation of the weak reference is persisted.

8.3.1.2.5 Weak Object Reference Vectors

[TBS.]

8.3.1.2.6 Weak Object Reference Sets

[TBS.]

8.3.1.3 Non-Persistent Data

Some attributes of persistent objects are not themselves persistent. Such attributes exist only to support in memory access to persistent objects. Examples are

1. Class dictionary entries for predefined classes
2. Object contexts
3. Those portions of the Object Directory that manage in-memory objects

8.3.2 How Save Works

When an object is in the persistent store a record of its location in persistent store is maintained in the object directory, a per-AAF file data structure. The object directory maps object references to the locations in persistent store of the referenced objects. When a reference is followed during a save operation the object directory is consulted first. If the referenced object is found in the object directory then its location in persistent store is returned and written to the persistent store. If the object is not found in the object directory then a new entry is made in the object directory and the object is written to persistent store by applying these rules recursively.

8.3.3 Saving a Single Object

This section will describe the steps performed by the *Object Manager* when saving a single object. This code exists only on class OMStorable since it is the same for all classes.

```
void OMStorable::save(void) const
{
  store()->save(classId());
  store()->save(_persistentProperties);
}
```

8.3.4 How Restore Works

When an object is in memory a record of its location in memory is maintained in the object directory, a per-AAF file data structure. The object directory maps object references to the locations in memory of the referenced objects. When a reference is followed during a restore operation the object directory is consulted first. If the referenced object is found in the object directory then its location in memory is returned as the result of the dereference operation. If the object is not found in the object directory then the Class Factory is used to create an uninitialized object instance of the appropriate sub-class. The location in memory of the newly created instance is entered into the object directory and the object is initialized from persistent store by applying these rules recursively.

8.3.5 Restoring a Single Object

This section describes the steps performed by the *Object Manager* when restoring a single object. This code exists only on class OMStorable since it is the same for all classes.

```
OMStorable* OMStorable::restoreFrom(const OMStorable* containingObject,
                                    const char* name,
                                    OMStoredObject& s)
{
  OMClassId cid;
  s.restore(cid);

  OMFile* f = containingObject->file();
  OMStorable* object = f->classFactory()->create(cid);

  object->setContainingObject(containingObject);
  object->setName(name);
  object->setStore(&s);

  f->objectDirectory()->insert(object->pathName(), object);

  object->restoreContentsFrom(s);

  return object;
}
```

8.3.6 Persisting References

8.3.6.1 Isomorphism

Supporting isomorphism means that

- Multiple in-memory references to the same in-memory object must be persisted as on-disk references to the same on-disk object.
- Multiple on-disk reference to the same on-disk object must be restored as in-memory references to the same in-memory object.

This is implemented by consulting the object directory during the save and restore operations.

- During a save operation the object directory is consulted to see if a given object instance has already been saved and, if so, the object is not saved again.
- During a restore operation the object directory is consulted to see if a given object instance has already been restored and, if so, the object is not restored again.

8.3.6.2 Circular References

Circular references are legal and the requirement for isomorphism means that they must be saved and restored. A persistence implementation must not recur or loop infinitely when faced with a request to persist an object network containing circular references. Persistence of circular references is implemented through the object directory, which ensures that exactly one reference is followed when a given object is persisted, thus breaking the cycle.

8.3.6.3 Null References

It is legal for both strong and weak references to be null references. Null references are handled as a generalization of object references. They are treated as if they were a reference to a fictional “null object”. During save each reference is checked to see if it is a null reference and if so it is persisted as a reference to the “null object”. During restore references to the null “object” are restored as null references.

8.4 Optional Properties

8.4.1 Data Manager View of Optional Properties

8.4.1.1 OMProperty Routines

```
class OMProperty {
public:
  ...

    // @cmember Is this an optional property ?
    //   @this const
  bool isOptional(void) const;

    // @cmember Is this optional property present ?
    //   @this const
  bool isPresent(void) const;

    // @cmember Remove this optional property.
  void remove(void);

  ...

};
```

8.4.1.2 Routine semantics

1. A property is either optional, `isOptional() == true`, or required, `isOptional() == false`.
2. An optional property may be present, `isPresent() == true`, or absent, `isPresent() == false`.
3. It is a programming error to ask if a required property is present, since, by very definition, a required property must be present.
4. An optional property may be removed, by calling `remove()`.
5. It is a programming error to attempt to remove an optional property that is not present. It is also a programming error to attempt to remove a required property.
6. Removing an optional property removes the value of that property from the in-memory and the on-disk representations of the object that contains the property. Once a property has been removed the property value is lost since the only way to make the property present again is to set the property to a new value thus destroying the old value,
7. Setting the value of an absent optional property makes the property present.

The routine pre and post conditions are summarized, more formally, as follows...

Routine	precondition	postcondition

`IsOptional()`	none	none
`IsPresent()`	`isOptional()`	none
`Remove()`	`isPresent()`	`!isPresent()`
get property value	`isOptional()` implies `isPresent()`	none
set property value	none	`isPresent()`

8.4.1.3 Validity Constraints

1) The concept of optionality is orthogonal to the concept of property value. Property values are subject to their own independent set of validity constraints. In particular
   a) A void strong reference property is different than an absent optional strong reference property.
   b) A strong reference vector property with no elements (or with all void elements) is different than an absent optional strong reference vector property.
   c) A null GUID is different than an absent weak reference property.
2) It is, however, a programming error to attempt to remove an optional property that represents valid containment. The DM must detach any contained objects before removing the optional property that contains them. The cases are
   a) A non-void OMStrongReferenceProperty.
   b) An OMStrongReferenceVectorProperty containing any non-void elements.

8.4.1.4 Declaring an Optional Property

There is no declaritive interface for optionality. Optional properties must be declared in the same way as required properties. The optionality of a property is defined when its containing object is created by `OMClassFactory::create()`. The `OMClassFactory` interface is defined by and used by the *Object Manager*, it is implemented by the *Data Manager* class `ImplAAFDictionary` using `AAFMetaDictionary.h`)

8.4.1.5 Accessing an Optional Property

When the *Data Manager* sets the value of an optional property that is not present the *Object Manager* makes the property present and sets it to the given value.

When reading an optional property the *Data Manager* must first check that the property is present. If the property is not present the *Data Manager* may choose either to return a default value for the property (as illustrated in the example) or to return a "property not present" error code.

```
class ExampleBase {
public:
  ...
  virtual void setOptionalInteger32(const OMUInt32& i);
  virtual void getOptionalInteger32(OMUInt32& i);
  ...
private:
  ...
  OMFixedSizeProperty<OMUInt32>           _optionalInteger32;
};

void ExampleBase::setOptionalInteger32(const OMUInt32& i)
{
  _optionalInteger32 = i;
}

void ExampleBase::getOptionalInteger32(OMUInt32& i)
{
  // If the optional property is present then return its value
  // otherwise return a default value.
```

```
    if (_optionalInteger32.isPresent()) {
      i = _optionalInteger32;
    } else {
      i = 0;
    }
  }
```

8.4.1.6 Removing a Simple Optional Property

Removing the `_optionalInteger32` property is simply

```
  _optionalInteger32.remove();
```

8.4.1.7 Removing Optional Containment

Given the following property declaration…

```
OMStrongReferenceProperty<ImplAAFFoo> _optionalFoo;
```

The property may be removed by…

```
ImplAAFFoo* oldFoo = _optionalFoo.setValue(0);
_optionalFoo.remove();
if (oldFoo != 0)
  oldFoo->ReleaseReference();
```

8.4.1.8 On-Disk Implications

An optional property that is not present for a given object instance has no entry in the property index for that object. That is, there is no on disk overhead for optional properties that are not present.

8.4.2 Validation

When checking objects for validity on save and restore, optional properties are, of course, permitted to be absent from the property index.

8.4.3 Internals

On class OMProperty there are the following state variables

```
bool _isOptional;
bool _isPresent;
```

The class invariant includes

```
INVARIANT("Mandatory property present", IMPLIES(!isOptional(), isPresent()));
```

The `OMProperty` instance representing a given optional property is always contained within the associated `OMPropertySet` whether or not the associated property value is present. The `OMPropertySet` represents the set of potential properties. Each `OMProperty` instance records, through the `_isPresent` state variable, whether or not the property value is present.
The `_isOptional` state variable is set at object creation time and cannot be changed.
The `_isPresent` state variable is set whenever the property value is set (either by restore from disk or via the `OMProperty` interface), The `_isPresent` state variable is reset when `remove()` is called and when, during restore, the on-disk property index is found not to include that property.

8.4.4 Dictionary

The dictionary (class ImplAAFDictionary), which implements `OMClassFactory::create`(), is responsible for properly initializing the properties of each newly created object according to the contents of `AAFMetaDictionary.h`.

8.5 Mapping of AAF Objects to Structured Storage

The basic mapping of AAF Objects to structured storage is as follows

- All AAF objects map to IStorages.
- Contained property values are stored in a single contained IStream.
- Contained AAF objects map to contained IStorages.

The IStorages and IStreams within a structured storage file form a tree. However, the objects associated with an AAF file form a network. This apparent clash of structures is resolved in this design by requiring that every object in the network, except the root object, have exactly one owner. That is, each object is contained within exactly one other object. This relationship is implemented by strong object references. The number of strong references to an object must always be one. When these conditions are met, the strong object references form a tree that includes all objects. This tree maps directly on to the tree formed by the IStorages and IStreams within a structured storage file.

Thus strong references are represented explicitly in memory and implicitly in the structured storage file.

Other, non-containing, non-owning, object associations are represented by weak references. There may be zero or more weak references to an object. Weak references are represented explicitly in memory and explicitly in the structured storage file.

8.5.1 Details of Mapping

1) Each AAF object is represented by a corresponding IStorage object. The class id of the AAF object is part of the IStorage object and is part of the structured storage overhead.
2) Each IStorage contains an IStream called "property index" that describes the contents of the "property values" IStream. The "property index" IStream contains a header followed by a counted array of structures.
   a) The header has the format.
      i) Byte order. Legal values are ‘II’ = Intel (little-endian), ‘MM’ = Motorola (big-endian).
      ii) Count of properties. The number of array elements that follow.
   b) The counted array has the format with the following fields
      i) Property id – an AUID that identifies this property.
      ii) Property stored form - the structural “type” of the property. This indicates the meaning of the information in the property values IStream. The valid property stored forms are
         (1) SF_DATA
         (2) SF_DATA_STREAM
         (3) SF_STRONG_OBJECT_REFERENCE
         (4) SF_STRONG_OBJECT_REFERENCE_VECTOR
         (5) SF_STRONG_OBJECT_REFERENCE_SET
         (6) SF_WEAK_OBJECT_REFERENCE
         (7) SF_WEAK_OBJECT_REFERENCE_VECTOR
         (8) SF_WEAK_OBJECT_REFERENCE_SET
         (9) SF_WEAK_OBJECT_REFERENCE_STORED_OBJECT_ID
         (10) SF_UNIQUE_OBJECT_ID
         (11) SF_OPAQUE_STREAM
         (12) SF_UNIQUE_STRONG_OBJECT_REFERENCE_VECTOR
      iii) Offset - the offset of the value of this property into the "property values” IStream.
      iv) Length – the length of the value of this property in the “property values” IStream.
3) Each IStorage contains an IStream called "property values" containing the properties for this object. The “property values” IStream consists of a sequence of property values.
4) Each contained object is stored in a sub-IStorage. The name of the IStorage is given by the value of an entry in the "property values" IStream with a stored form of SF_STRONG_OBJECT_REFERENCE
5) Contained vectors of objects are represented as follows

a) Each collection is described by an IStream with a name given by the value of an entry in the “property values” IStream with a stored form of SF_STRONG_OBJECT_REFERENCE_VECTOR, if the value is "foo", the stream is named “foo index”.
b) The content of this IStream is a counted array of local keys (integers), and a “high water mark”, indicating the lowest unused local key.
c) Each local key is used to construct the name of the sub-IStorage corresponding to the object at that position in the collection. If the value of the first local key in the array is 42 then the name of the sub-IStorage used to store the first object in the collection is "foo{42}". The local keys are assigned in a non-repeating ascending sequence, using the “high water mark”. The sequence of local keys is specific to this collection. Local keys are assigned in this fashion to avoid having to rename any IStorages when elements are inserted into or removed from the collection.

6) Contained sets of objects are represented as follows. [TBS.]
7) Inter-object references are represented as follows
   a) The value of an inter-object reference is the AUID of the referenced object.
   b) Value corresponds to an entry in the "property values" IStream with a stored form of SF_WEAK_OBJECT_REFERENCE.
8) Vectors of inter-object references are represented as follows. [TBS.]
9) Sets of inter-object references are represented as follows. [TBS.]
10) Properties that are media data are represented as follows
   a) The "value" of a media data property is the name of a sub-IStream containing that data.
   b) The value corresponds to an entry in the "property values" IStream with a stored form of SF_DATA_STREAM.
11) The unique strong reference set and vector contain uniquely identified objects that may be the target of a weak reference.
12) Sets provide an efficient (binary tree search) lookup of object by guid. This is used to find a definition in the dictionary given the AUID of the definition, and to find a Mob given a MobId.

8.5.2 Examples

8.5.2.1 An Instance of AAFSequence

Parent Storage (contains AUID_AAFSequence)

Stream "property index"

Header

```
XX    // byte ordering
X     // version number of this representation
4     // count of properties
```

Index

Property (PID_)	Stored Form (SF_)	Offset	Length
Component_DataDefinition	DATA	00	16
Component_Length	DATA	16	08
Sequence_Components	STRONG_OBJECT_REFERENCE_VECTOR	24	11

Stream "property values"

Value

```
xxxxxxxxxxxxxxxx    // AUID      AAFComponent::DataKind
xxxxxxxx            // INT64     AAFComponent::Length
"components"        // STRING    AAFSequence::Components
```

Stream "components index"

```
43          // high water mark
02          // count of elements
42          // local key of first element
13          // local key of second (and last) element
```

Storage "components 42"

An instance of AAFComponent (or sub-class)

Storage "components 13"

An instance of AAFComponent (or sub-class)

8.5.2.2 Example Dump

This example is the dump of an AAFSequence object (similar to the one above) by a low-level dump program.

```
/Content/Mobs{0}/Slots{0}/Segment
Dump of property index
( Byte order = little endian (native), Version = 7, Number of entries = 3 )
    property   pid (hex)        type      offset      length
           0         201           0           0          16
           1         202           0          16           8
           2        1001           2          24          11

Dump of properties

property 0 ( data )
       0    e1 eb e1 78 ef 6c d2 11 80 7d 00 60 08 14 3e 6f    ...x.l...}.`..>o

property 1 ( data )
       0    32 00 00 00 00 00 00 00                            2.......

property 2 ( strong object reference vector )
       0    43 6f 6d 70 6f 6e 65 6e 74 73 00                   Components.

/Content/Mobs{0}/Slots{0}/Segment/Components
Dump of vector index
( High water mark = 5, Number of entries = 5 )
 ordinal    local key
       0 :          0
       1 :          1
       2 :          2
       3 :          3
       4 :          4
```

8.5.2.3 Example Dump of a Set Index

```
Dump of set index
( High water mark = 3, Number of entries = 3 )
 ordinal    local key   references     unique key
       0 :          0            1     {0D1EDA00-7752-11D3-801D-080036210804}
       1 :          1            1     {0D1EDA01-7752-11D3-801D-080036210804}
       2 :          2            1     {0D1EDA02-7752-11D3-801D-080036210804}
```

8.5.2.4 [Other Examples TBS.]

[TBS. As other examples of how AAF objects are mapped to structured storage are created they will be added here.]

8.5.3 Data Structures

This section describes the data structures used to map AAF object on to structure storage. Note that these are not the actual data structures, they are provided for illustrative purposes only.

8.5.3.1 Integral Types

These types, assumed to be defined appropriately for a particular host, are used in subsequent declarations.

```
typedef … UInt8;
typedef … UInt16;
typedef … UInt32;
```

8.5.3.2 Property Index

8.5.3.2.1 Purpose

The property index is an index into the property values in the property values stream.

8.5.3.2.2 External representation

An IStream called “property values” containing a `PropertyIndexHeader` followed by `_entryCount` `PropertyIndexEntry` structs.

8.5.3.2.3 Structure of Property Index Header

A `PropertyIndexHeader` is defined as follows…

```
typedef struct PropertyIndexHeader {
  UInt16 _byteOrder;
  UInt32 _formatVersion;
  UInt32 _entryCount;
} PropertyIndexHeader;
```

The `_byteOrder` is the byte order of

- the remaining fields of the `PropertyIndexHeader` struct
- the `PropertyIndexEntry` structs that follow
- the actual property data

The `_formatVersion` is version number of the stored format, this allows for otherwise incompatible changes to the stored format.
The `_entryCount` is the number of `PropertyIndexEntry` structs that follow.

8.5.3.2.4 Structure of a Property Index Entry

```
typedef struct PropertyIndexEntry {
  UInt32 _pid;
  UInt32 _storedForm;
  UInt32 _offset;
  UInt32 _length;
} PropertyIndexEntry;
```

The `_pid` is the id that describes the property. This is a shorthand version of the AUID that uniquely identifies the property. Property ids are locally unique. For all predefined AAF properties the property id is the same in all AAF files. For user defined extension properties the assigned property id may vary across files.
The `_storedForm` identifies the “type” of representation chosen for this property. This field describes how the property value should be interpreted. Note that the stored form described here is not the data type of the property value, rather it is the type of external representation employed. The data type of a given property value is implied by the property ID. The actual data type of a property value may be determined by looking up the associated property id in the AAFDictionary.
The `_offset` is the byte offset of the property value in the property value stream.
The `_length` is the length, in bytes, of the property value in the property value stream.

8.5.3.3 Strong Object Reference

8.5.3.3.1 Purpose

A single contained object.

8.5.3.3.2 External Representation

Stored form	SF_STRONG_OBJECT_REFERENCE
Property value	name of object

8.5.3.3.3 Structure of a Strong Object Reference

[TBS]

8.5.3.4 Strong Object Reference Vector

8.5.3.4.1 Purpose

An ordered collection of strongly referenced (contained) objects.

8.5.3.4.2 External Representation

Stored form	SF_STRONG_OBJECT_REFERENCE_VECTOR
Property value	name of vector
Set index name	<name of vector> index
Set element name	<name of vector>{<local key of element>}

Each vector index consists of a `StrongReferenceVectorIndexHeader` followed by `_entryCount` `StrongReferenceVectorIndexEntry` structs.

8.5.3.4.3 Structure of a Strong Object Reference Vector Index Header

A `StrongReferenceVectorIndexHeader` is defined as follows…

```
typedef struct StrongReferenceVectorIndexHeader {
  UInt32 _entryCount;
  UInt32 _highWaterMark;
} StrongReferenceVectorIndexHeader;
```

The `_highWaterMark` is the highest local key ever assigned to an element of this vector. It is one less than the next local key that will be assigned in this vector.
The `_entryCount` is the number of `VectorIndexEntry` structs that follow.

8.5.3.4.4 Structure of a Strong Object Reference Vector Index Entry

```
typedef struct StrongReferenceVectorIndexEntry {
  UInt32 _localKey;
} StrongReferenceVectorIndexEntry;
```

The `_localKey` uniquely identifies this strong reference within this collection independently of its position within this collection. The `_localKey` is used to form the name assigned to the element in this vector at the corrseponding ordinal position. That is, the `_localKey` of the first `StrongReferenceVectorIndexEntry` is used to form the name of the first element in the vector and so on. The `_localKey` is an insertion key.

8.5.3.5 Strong Object Reference Sets

8.5.3.5.1 Purpose

An unordered collection of strongly referenced (contained) uniquely identified objects, each of which can be

- efficiently located by key - O(lg N)
- the target of a weak reference

8.5.3.5.2 External Representation

Search key	obtained from "object->identifier()"
Stored form	SF_STRONG_OBJECT_REFERENCE_SET
Property value	name of set
Set index name	<name of set> index
Set element name	<name of set>{<local key of element>}

StrongReferenceSetIndexEntry structs appear in the index in order of increasing key. If an application consuming the set index wishes to construct a binary search tree, care must be taken not to invoke the worst case performance by inserting the keys in order. One way to avoid this problem is to insert the keys in "binary search" order. That is the middle key is inserted first then (recursively) all the keys below the middle key followed by (recursively) all the keys above the middle key.
Each set index consists of a `StrongReferenceSetIndexHeader` followed by `_entryCount StronReferenceSetIndexEntry` structs.

8.5.3.5.3 Structure of a Strong Object Reference Set Index Header

```
typedef struct StrongReferenceSetIndexHeader {
  UInt32 _entryCount;
  UInt32 _highWaterMark;
  UInt32 _identificationPid;
  UInt32 _identificationSize;
} StrongReferenceSetIndexHeader;
```

The `_identification` field of `StronReferenceSetIndexEntry` is the value of the property on the contained objects with property id `_identificationPid.` Each `_identification` in the `StrongReferenceSetIndexEntry` structs that follows is `_identificationSize` bytes in size.

8.5.3.5.4 Structure of a Strong Object Reference Set Index Entry

```
typedef struct StrongReferenceSetIndexEntry {
  UInt32     _localKey;
  UInt32     _referenceCount;
  <variable> _identification;
} StrongReferenceSetIndexEntry;
```

The `_referenceCount` is the count of weak references to this object. The type of the `_identification` field varies from one instance of a StrongReferenceSet to another. The value of the `_identification` field uniquely identifies this object within the set. It is the search key.

8.5.3.6 Weak Object Reference

8.5.3.6.1 Purpose

A weak object reference is a persistent data type that denotes a weak reference to a uniquely identified object. In memory, weak references are similar to pointers. When persisted, weak references contain the unique identifier of the referenced object.

8.5.3.6.2 External representation

Stored form	SF_WEAK_OBJECT_REFERENCE

8.5.3.6.3 Structure of a Weak Object Reference

```
typedef struct WeakObjectReference {
  UInt32     _referencedPropertyIndex;
  UInt32     _identificationPid;
  UInt32     _identificationSize;
  <variable> _identification;
} WeakObjectReference;
```

The `_referencedPropertyIndex` is the index into the referenced property table of the name of the property (a strong reference set) containing the referenced object. The type of the `_identification` field varies from one

instance of a `WeakObjectReference` to another. The `_identification` field uniquely identifies the object within the target set.

8.5.3.7 Weak Object Reference Vector

8.5.3.7.1 Purpose

An ordered collection of weak references.

8.5.3.7.2 External representation

Stored Form	SF_WEAK_OBJECT_REFERENCE_VECTOR
Property value	name of vector
Vector index name	<name of vector> index

8.5.3.7.3 Structure of a Weak Reference Vector Index Header

```
typedef struct WeakReferenceVectorIndexHeader {
  UInt32 _entryCount;
  UInt32 _referencedPropertyIndex;
  UInt32 _identificationPid;
  UInt32 _identificationSize;
} WeakReferenceVectorIndexHeader;
```

8.5.3.7.4 Structure of a Weak Object Reference Vector Index Entry

```
typedef struct WeakReferenceVectorIndexEntry {
  <variable> _identification;
} WeakReferenceVectorIndexEntry;
```

8.5.3.8 Weak Object Reference Set

8.5.3.8.1 Purpose

An unordered collection of weakly referenced (not contained) uniquely identified objects, each of which can be

- efficiently located by key - O(lg N)

8.5.3.8.2 External Representation

Search key	obtained from "object->identifier()"
Stored form	SF_WEAK_OBJECT_REFERENCE_SET
Property value	name of set
Set index name	<name of set> index

8.5.3.8.3 Structure of a Weak Object Reference Set Index Header

```
typedef struct WeakReferenceSetIndexHeader {
  ... same as WeakReferenceVectorIndexHeader ...
} WeakReferenceSetIndexHeader;
```

8.5.3.8.4 Structure of a Weak Object Reference Set Index Entry

```
typedef struct WeakReferenceSetIndexEntry {
  ... same as WeakReferenceVectorIndexEntry ...
} WeakReferenceSetIndexEntry;
```

8.5.3.9 Stored Object Identification

8.5.3.9.1 Purpose

The purpose of this stored form is to
- avoid storing an additional copy of information. The stored object identification is logically a property but is physically stored as the IStorage class identifier for the IStorage that represents the object.
- treat object properties uniformly

The property value is the "IStorage class identifier". This value is set using IStorage::SetClass() and obtained with IStorage::Stat().
The meaning of a stored object identification (SF_WEAK_REFERENCE_STORED_OBJECT_ID) is the same as that of a weak reference (SF_WEAK_OBJECT_REFERENCE) except that the unique identifier of the referenced object (The defining instance of ClassDefintion) is persisted differently.

8.5.3.9.2 External representation

Stored Form	SF_WEAK_REFERENCE_STORED_OBJECT_ID

8.5.3.10 Unique Object Identification

8.5.3.10.1 Purpose

The purpose of this stored form is to
- avoid storing an additional copy of information. The unique object identification is logically a property but is physically stored in the index of the collection of which the object is a member so that the collection may be searched without having to load objects.
- treat object properties uniformly

More [TBS.]

8.5.3.10.2 External Representation

Stored form	SF_UNIQUE_OBJECT_ID

8.5.3.11 Opaque Stream

8.5.3.11.1 Purpose

The property value is a small (size is such that fit easily into memory) data stream. The contents of the data stream are opaque to the Object Manager. Items of this type are not a part of the AAF object model. This stored form is intended for use by Object Manager clients in defining such "standard" Structured Storage elements as the "DocumentSummaryInformation" stream.

8.5.3.11.2 External Representation

Stored Form	SF_OPAQUE_STREAM

8.5.4 The Referenced-Properties Table

A weak object reference consists of
- the AUID that uniquely identifies the referenced object
- the name of the property that contains the referenced object.

The reference contains the name of the property that contains the referenced object in order to avoid having to search through all uniquely identified objects or having to build a data structure whose size scales linearly with the number of weakly referenced objects to support resolution of weak references.

In order to avoid storing the actual name of the referenced property in each weak reference the name is stored once, in the referenced-properties table, and the index of the name in the table is stored in the weak reference.
There is one referenced-properties table in each AAF file. The referenced-properties table is a stream called "`/referenced rroperties`". The stream consists of a header followed by a string space. The `_size` field of the header gives the size in bytes of the string space. The total size of the referenced-properties stream is `_size + sizeof(ReferencedPropertiesTableHeader)`, that is, `_size + 16` bytes.

8.5.4.1 The Referenced-Properties Table Header

```
typedef struct ReferencedPropertiesTableHeader {
  UInt32 _count;
  UInt32 _size;
} ReferencedPropertiesTableHeader;
```

The `_count` field holds the number of referenced-properties in the table. The `_size` field is the total size, in bytes, of the string space that follows.

8.5.4.2 The Referenced-Properties Table String Space

The referenced-properties table string space is a sequence of null terminated characters strings each string names a referenced property. The first string in the string space has index 0 in the referenced-properties table and so on.

8.5.4.3 The Referenced-Properties Table Validity constraints

The string space must contain exactly `_count` bytes that have the value 0. The length of the referenced-properties stream must be `_size` + 16. Each of the strings in the string space must be unique. That is, no two strings may be the same. Each of the strings in the string space must be used by some weak reference.

8.5.4.4 Scalability of the Referenced-Properties Table.

There are as many entries in the referenced-properties table as there are properties that contain weakly referenced objects. For example, there is only one entry in the referenced-properties table for `/Dictionary/ClassDefinitions` even though there are many weak references to class definitions.

8.5.5 General Design Principles

1) The value of a weak reference is a pointer to the referenced object
2) An object may be loaded when either
   a) the one and only strong reference to the object is followed
   b) any weak reference to the object is followed
3) Only certain classes may be weakly referenced. Such objects are uniquely identified by UID. The unique identifier is used (as a key) to
   a) identify the object that is the target of a weak reference
   b) identify the object within the collection in which it resides
   c) identify the object within internal Object Manager data structures
4) By definition set elements are uniquely identified.
5) Elements in a strong reference set and in a unique strong reference vector are unique within their respective containers by the definition of strong reference - there can be only one strong reference to a given object. This is equivalent to "pointer identity".
6) Elements in a strong reference set and in a unique strong reference vector are unique within their respective containers by GUID. That is, the GUID is used as a key.
7) There is no SF_UNIQUE_WEAK_OBJECT_REFERENCE_VECTOR since weak references may not themselves be the target of other weak references.
8) Weakly referencable objects have persisted reference counts. The reference count is persisted so that the reachability (or liveness) of a given object may be determined without having to load objects to find all references
9) The reference count is persisted in such a way that it can be manipulated without loading the object
10) Only objects with a non-zero weak reference count are persisted. This is the basis of implementing notions such as "the dictionary contains only those definitions that are used in the file."
11) All of the weak references within the same collection (weak reference set or weak reference vector) refer to objects in the same target collection (strong reference set or unique strong reference vector)

12) A mechanism will be provided to allow uniform iteration over all collections set/vector, strong/weak unique/not unique
13) For uniquely identified objects the unique identifier is persisted separately from the object that it identifies. This allows
    a) building a binary search tree of objects without actually loading the objects from the file
    b) determining if an object is present in a collection without causing objects to be loaded
14) There is no SF_UNIQUE_STRONG_REFERENCE since, in the current AAF object model there is no case in which a single contained object is the target of a weak reference.
15) There is no SF_UNIQUE_STRONG_OBJECT_REFERENCE_SET since by definition the members of a set are unique.

8.5.6 Extra Design Flexibility

This section describes some extra flexibility built into the low-level AAF stored format/*Object Manager* design and that may be exploited in the future. This extra flexibility comes at a relatively small incremental (space and time) cost and may provide some extra room for maneuver later.

8.5.6.1 Per-Object Byte Order

Although the AAF requirements state that all objects in a file are stored with the same byte order, the byte order is specified on a per-object basis. This means that AAF could allow files containing a mixture of objects with different byte orders if needed. This could make for faster edit in place of foreign files.

8.5.6.2 Per-Object Format Version

The stored format is versioned and the version is specified on a per-object basis. This means that objects of different stored format versions could be allowed in the same file. This may have pay-back when, for example, both AAF version 1.0 and AAF version 2.0 are in the field and a read-modify-write operation is performed on a version 1.0 file using a version 2.0 implementation. In this case the unmodified version 1.0 objects in the file would not have to be converted to version 2.0 format.

8.5.7 Meta-data Byte Order

Although the AAF requirements for file byte order need not apply to the *Object Manager* meta-data, this design chooses to apply the same rules. These are, in summary
1) When a file is created, the meta-data within it is created with the byte order of the host
2) When a file is modified the existing byte ordering of the meta-data is preserved

One consequence of these rules is that all property indexes in a given file have the same byte order.

8.5.8 Storage Overhead

There are currently the following categories of property each having a different mapping to structured storage.

Property Stored Form	Meaning
SF_DATA	an ordinary property
SF_STRONG_OBJECT_REFERENCE	a contained object
SF_STRONG_OBJECT_REFERENCE_VECTOR	a vector of contained objects
SF_WEAK_OBJECT_REFERENCE	a reference to an object
SF_DATA_STREAM	media data
SF_WEAK_OBJECT_REFERENCE_VECTOR	a vector of references to objects

8.5.8.1 General storage overhead

1) every object consists of
    a) an IStorage
    b) an IStream for all of the SF_DATA properties in that object called "property values"
    c) an IStream for the property index called "property index"

2) each property index has a fixed overhead of 10 bytes for the index header
   a) byte order = 2 bytes
   b) version = 4 bytes
   c) number of properties = 4 bytes
3) each property index entry is 16 bytes in size
   a) property id (identifies the property within the class of this object) = 4 bytes
   b) property stored form (SF_ value) = 4 bytes
   c) offset of the property value in the "property values" stream = 4 bytes
   d) length of the property value in the "property values" stream = 4 bytes
4) each SF_STRONG_OBJECT_REFERENCE_VECTOR has an overhead of
   a) an IStream for the vector index
5) each vector index has a fixed overhead of 8 bytes for the index header
   a) high water mark = 4 bytes
   b) number of elements= 4 bytes
6) each vector index entry is 4 byes in size
   a) local key of element= 4 bytes
7) each SF_STRONG_OBJECT_REFERENCE_SET has an overhead of
   a) an IStream for the set index
8) each set index has a fixed overhead of 8 bytes for the index header
   a) high water mark = 4 bytes
   b) number of elements= 4 bytes
9) each set index entry is 4 byes in size
   a) local key of element= 4 bytes
   b) unique identifier (key) of element (an AUID) = 16 bytes

8.5.8.2 Storage overhead for each property category

Object Manager Stored Form	Overhead
SF_DATA	one property index entry
SF_STRONG_OBJECT_REFERENCE	one property index entry
SF_STRONG_OBJECT_REFERENCE_VECTOR	one property index entry + one vector index
SF_WEAK_OBJECT_REFERENCE	one property index entry
SF_DATA_STREAM	one property index entry + one IStream
SF_WEAK_OBJECT_REFERENCE_VECTOR	TBD (will probably be one AUID, 16 bytes per reference)

8.5.8.3 Some formulas

Size of property index stream = 10 + (number of properties * 16)
Size of vector index.= 8 + (number of elements * 4)

8.5.8.4 Storage Optimizations

There are two storage optimizations planned for the *Object Manager* not listed above. The optimizations are

1) If a property value is smaller than the index overhead for the property value then the property value is stored in the index itself. Property values stored this way are called immediate values. An index entry is 16 bytes of which 8 bytes (4 bytes for the property id and 4 bytes for the property stored from) are still needed for immediate values. This means property values 8 bytes in size or smaller may be stored as immediate values. This optimization is only possible for property values whose size is implied by their type. As a consequence, this optimization is only possible for fixed size property values. A property stored form of SF_IMMEDIATE_DATA identifies immediate values. If all the properties of an object are immediate there is no "property values" stream only a "property index stream".
2) If the sum of the size of the "property values" stream and the size of the "property index" stream is less than the minimum stream size then both the property index and the property values are stored in the same stream. The minimum stream size is reportedly 128 bytes. The stream is organized such that the property values follow the property index. Such a combined stream is named "property index" in which case there is no "property values"

stream. This means that all AAF file consumers may begin by opening the "property index" stream, which will always be present.

These two optimizations, which may of course be combined in the same object, will result in a best case object representation of one IStorage and one IStream.

We choose not to make these optimizations mandatory in order to support "edit in place" in which case changes might cause an objects representation to flip in and out of the optimized state. Instead we make the optimizations optional and provide separate mechanisms for casting an object in its canonical form.

Since these optimizations are not mandatory they introduce the possibility of alternate valid representations for the same object. (This possibility is also introduced during "edit in place" as such modifications may introduce garbage). Object can be transformed to their canonical representation by copy() or move() operations.

In addition, the sizes of the fields in the data structures listed in section 8.4.3 are subject to change during development based on measurements of real AAF files. For example the current definitions allow a 4Gb maximum total size of property data (not counting media data) on a single object; this may be excessive.

8.5.9 Property Ids

The general form of a property id name is PID_<ClassName>_<PropertyName>. Some examples of property ids are -

1) PID_Component_DataDefinition
2) PID_Component_Length
3) PID_Sequence_Components

8.5.10 Stored Class Ids

This note documents the mapping between SMPTE unique identifiers and AUIDs.
SMPTE has allocated a portion of a 16 byte namespace to AAF. The octets (bytes) are numbered from most to least significant. Identifiers from this namespace are of the following form ...

	SMPTE identifier															
Octet #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Octet Value	06	0E	2B	34	01	01	01	XX	XX	XX	XX	XX	XX	XX	XX	XX

where XX denotes octets within the namespace that the definers of AAF are free to allocate. These octets have been allocated in the spreadsheet as follows.

	SMPTE identifier															
Octet #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Octet Value	06	0E	2B	34	01	01	01	$A	$B	$C	$D	$E	$F	$G	$H	$I

Where $A - $I represent the spreadsheet columns A - I. Note that spreadsheet column A is always 04.

To transform this into a AUID that we know won't collide with any other GUID we simply exchange octets 0-7 with octets 8-15. This works because GUIDs with the most significant bit of octet 8 set to 0 are reserved but will never be allocated by the body that reserved them ! This gives ...

	AUID															
Octet #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Octet Value	$B	$C	$D	$E	$F	$G	$H	$I	06	0E	2B	34	01	01	01	$A

Given that $A is always 04 this results in the following mapping from the spreadsheet to an AUID ...

	AUID															
Octet #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Octet Value	$B	$C	$D	$E	$F	$G	$H	$I	06	0E	2B	34	01	01	01	04

An AUID is defined using the `DEFINE_AUID()` macro as follows ($A - $I represent spreadsheet columns A-I)…

```
DEFINE_AUID(name,
            0x$B$C$D$E,
            0x$F$G, 0x$H$I,
            0x06, 0x0E, 0x2B, 0x34, 0x01, 0x01, 0x01, 0x$A)
```

8.5.10.1 Example

This example shows the spread sheet entry, the SMPTE identifier, the AUID and the initialization code for the class AAFIdentification.

	Spread sheet entry for AAFIdentification								
Spreadsheet column	$A	$B	$C	$D	$E	$F	$G	$H	$I
Column value	04	06	49	00	00	00	00	00	00

	SMPTE identifier for AAFIdentification															
Octet #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Octet Value	06	0E	2B	34	01	01	01	04	06	49	00	00	00	00	00	00

	AUID for AAFIdentification															
Octet #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Octet Value	06	49	00	00	00	00	00	00	06	0E	2B	34	01	01	01	04

The AUID for the class AAFIdentification would be declared (or defined – depending on the currently effective definition of the macro `DEFINE_AUID`) as follows …

```
DEFINE_AUID(AUID_AAFIdentification,
            0x06490000,
            0x0000, 0x0000,
            0x06, 0x0E, 0x2B, 0x34, 0x01, 0x01, 0x01, 0x04)
```

8.5.11 Code class ids vs. Stored class ids.

8.5.11.1 Requirements and motivation

AAF admits the possibility of alternate implementations than the reference implementation. This means that the implementation of AAF that reads a file may not be the same implementation of AAF that wrote the file.
It is a requirement of interchange that AAF objects produced by different implementations be identified in the same way. In other words, an AAFSourceClip is always identified in an AAF file in the same way, regardless of which implementation of AAF created the file. An AAF object produced by one implementation should be identified in an AAF file in exactly the same way as a logically identical object produced by another implementation.
Since end users will purchase applications from different vendors that use different implementations of AAF and expect those applications to interoperate via AAF files we need to allow more than one implementation of AAF to exist on the same system.

8.5.11.2 Consequences

Taken together these factors result in a design in which the notion of a stored class id is different from the notion of a code class id.

8.5.11.2.1 Stored class ids

Stored class ids identify the class of an AAF object and, for a given AAF class, are the same across all implementations of AAF. Stored class ids are represented in the AAF object model by the ObjClass property of the AAFInterchangeObject class.

8.5.11.2.2 Code class ids

Code class ids identify a class from a particular implementation of AAF. Each implementation is free to choose its own mapping from stored class id to code class id when reading an AAF file (The simplest mapping is 1:1 with the implementation providing a different class to handle each stored class id). Note that code class ids are equivalent to COM class ids. COM class ids may be stored in a structured storage IStorage using IStorage::SetClass() and retrieved using IStorage::Stat(). Structured storage always allocates space in the IStorage for a code class id whether or not anything is actually stored there.

8.5.11.3 Design Details

The following sections detail the design as manifest in the stored format and in the reference implementation code.

8.5.11.3.1 Stored Format Design Details

a) Only stored class ids are stored in an AAF file.
b) Code class ids are never stored in an AAF file.
c) To conserve space the stored class ids are stored in the IStorage representing the stored object using IStorage::SetClass() and retrieved using IStorage::Stat(). i.e the proposal is to put the stored class id in the place already allocated but more usually used for code class ids.

8.5.11.3.2 Reference Implementation Code Design Details

a) Each AAF class is assigned a stored class id, these ids are the same for all AAF implementations.
b) Each AAF class in the reference implementation is assigned a code class id, these ids are specific to the reference implementation.
c) The stored class id is available from any AAF object via the AAFInterchangeObject::GetObjectClass() method.
d) When an object is persisted AAFInterchangeObject::GetObjectClass() (or equivalent) is called to obtain the objects stored class id. This stored class id is written to the structured storage IStorage respesenting the object using IStorage::SetClass().
e) When an object is "unpersisted" its stored class id is retrieved from the AAF file using IStorage::Stat(). The stored class id is mapped to a code class id (via the AAFDictionary) so that an object instance appropriate to the reference implementation can be created. Ultimately a call is made to CoCreateInstance() passing in the code class id.

8.5.11.4 Design Discussion

The following sections explore some some of the consequences of this design.

8.5.11.4.1 Putting the stored class id where the code class id should go

Although stored class ids are a distinct concept from code class ids the proposal is to store them in the place normally reserved for code class ids (COM class ids). This proposal saves 16 bytes per object. There are some risks associated with this

1) the risk of confusing stored class ids with code class ids
2) an attempt may be made to call OLELoad() or CreateInstanceFromIStorage() on an IStorage in an AAF file. These should and will fail but will they fail with an error code different than if nothing at all were stored in the IStorage ?

8.5.11.4.2 Is This the Usual COM Practice?

It may be argued that this proposal is not in accordance with usual COM practice. That is true. However, this is not intrinsic to this proposal but instead follows from the need to allow alternative implementations of AAF.

[TBS quote from Microsoft dicumentation that says that the field in the IStorage is an “indication” of the owning class.]

8.5.11.5 Design Alternatives

The following design alternatives were considered.

8.5.11.5.1 Use an Explicit Property for the Stored Object Id

Don't put any id in the IStorage, instead represent the stored class id as an explicit property. This costs 16 bytes for the stored class id plus 16 bytes overhead in the property index. This also means that the code reading the file has to read the property index for an object before it can create that object. It also raises the issue of where in the index this "special" property is stored. The benefit of this is that stored class ids can't be as easily confused with code class ids since IStorage::Stat() will always return a null code class id.

8.5.11.5.2 Use a File Local Identifier

Don't store a guid at all, but instead save space by storing a small integer in each object (or in the IStorage) that is mapped to a stored class id. For extension classes that mapping is accomplished by a lookup table (or similar data structure) that is also stored in the file (as part of the dictionary).
For extension classes the small integer has meaning only within a given file. That is, a different small integer may be assigned if the object is copied to another file. Of course in the new file the newly assigned small integer will map to the same stored class id.
For predefined classes the mapping is not persisted (just as for all other predefined dictionary entries) and predefined classes are always assigned the same small integer.
This is essentially the same as my proposal for uniquely identifying properties without having to consume vast amounts of space by storing a guid with each property.

8.5.12 Canonical Forms

The definition of the stored format presented here allows multiple respresentations of the same object. The multiple representations arise for the following reasons –

1) There is no defined order for the properties of an object.
2) The property values in the property values stream need not be in the order specified in the property index.
3) The property values in the property values stream need not be contiguous. That is there may be gaps between the values. This could arise, for example, if a value has been edited in place and shortened leaving a gap.
4) An optimization allows the property index and the property values to be stored in the same stream. This optimization may or may not be applied to a particular object.
5) An optimization allows small property values to be stored in the property index. This optimization may or may not be applied to a particular object.

This specification does **not** impose a canonical or preferred representation of persisted AAF objects. The *Object Manager* must be prepared to accept all of the above variations on input. On output of a modified object the *Object Manager* is not required to alter the representation of unchanged properties in order to create a canonical representation of the object. When an object is first written to persistent store, for example as the result of a copy operation, the *Object Manager* will create the object using the most compact representation possible. This enables the creation of a simple application that reduces AAF files to their smallest possible size by simply copying all of the contained objects.

8.5.13 Garbage Collection

Since the *Object Manager* supports edit in place, and since no canonical representation of objects is imposed, garbage, or unused and unreachable space, may accumulate in the property value stream. Such unused and unreachable space is permitted in an AAF file. The garbage is reclaimed when the object is copied. Such reclamation would therefore also be achieved by the simple file compacting application mentioned above.

8.5.14 Using This Mapping to Implement IPersistStorage

When using COM/OLE an object must be completely initialized by a call to its implementation of IPersistStorage::Load. In this proposal the initial state for an object is found in more than a single IStorage. The AAF Class dictionary may also contain part of the initial state for an object. A utility function similar to OLELoad, say AAFLoad, could be provided. The implementation of AAFLoad would have to

1. Read the class id from the IStorage representing the persisted object.
2. Consult the AAF class dictionary to find the creation function to use to create objects of that class.
3. Call the creation function.
4. Call IPersistStorage::Load which would create a partially initialized instance of the object.
5. Call the newly created objects "initialize" method, passing in a reference to the class dictionary. This method would complete initialization of the object by referring to the class dictionary, for example, to turn type ids into the corresponding AAFDefinition objects.

Note that once AAFLoad has completed the object is not entirely loaded into memory since the loading of some properties may have been deferred because of lazy loading.

8.5.15 Storage and Stream Names

This section describes the naming convention adopted for naming IStorages and IStreams with in a structured storage file. The rules are as follows:

1) The root IStorage is called "/" and is the stored representation of the AAFHeader object.

2) An IStorage that represents a contained object is given the name of the property that it represents. For example, the AAFHeader object contains an AAFDictionary object with property name "Dictionary" so the name of the IStorage containing the stored representation of the AAFDictionary is "/Dictionary".

3) All objects also contain an index to their properties. This index is stored in an IStream called "property index". Continuing the example, the property index IStream for the AAFDictionary is called "/Dictionary/property index".

4) If an object contains properties that are not themselves objects then the values of those properties are stored in an IStream called "property values". Continuing the example, the property value IStream for the AAFDictionary is called "/Dictionary/property values"

5) If an object contains a strong reference vector (vector of contained objects) then the object will contain an index for the vector stored in an IStream called "<VectorPropertyName> index". Elements of the vector are stored IStorages called "<VectorPropertyName>{<element name>}". Continuing the example, since the AAFDictionary contains a strong reference property called "ClassDefinitions" consisting of AAFClassDefinition objects the vector index stream will be called "/Dictionary/ClassDefinitions index". Since each element in the vector is an object the vector elements will be stored in IStorages named "/Dictionary/ClassDefinitions{<element name>}". Where <element name> is a hexadecimal name that identifies the element and is chosen such that elements don't have to be renamed if insertions or deletions are made to the vector. A vector index is mapped to its corresponding name by the vector index. There is not currently a need to map from a name back to the corresponding index. However, this stored format design does not preclude adding this in the future.

6) If an object contains a restricted weak reference vector (vector of referenced objects). [TBS. This section will contain a description of the stored representation of restricted weak reference vectors.]

8.5.16 Storage of Object References and Object Reference Arrays

This section describes the details of how object references and object reference arrays are mapped to structured storage.

8.5.16.1 Strong References

Strong references denote containment. If an AAF object contains another AAF object the IStorage containing the stored representation of the containing object has the IStorage containing the stored representation of the contained object as a sub-storage.

8.5.16.2 Restricted Weak References

8.5.16.2.1 Restricted Weak References in the AAF Object Model

Instances of the following classes may be the target of weak references

1. Definition objects
   - DataDefinition
   - EffectDefinition
   - ClassDefinition
   - PropertyDefinition
   - TypeDefinition
2. Mobs
   - MasterMob
   - CompositionMob
   - SourceMob
3. Media
   - MediaData

8.5.16.2.2 Representation of Restricted Weak References

A weak reference to an object instance is stored as an AUID (GUID). To follow a weak reference the AUID is looked up in a per-class data structure that maps AUIDs onto object instances of that class. These data structures are part of the AAF object model and their persisted representation is described in the "Advanced Authoring Format Object Specification". For example, to find the Mob referenced by a given AUID that AUID must be looked up in a run-time data structure the persisted representation of which is "/Header/Content/Mobs".[1]

8.5.16.2.3 Implementation of Restricted Weak References

[TBS. This section will describe the implementation of restricted weak references.]

8.5.16.3 General Weak References

The AAF object model does not require general weak references. This design does, however, take them into account. General weak references could be added to this design in a straightforward manner if this becomes a future requirement. One possible approach to implementing general weak references would be to construct and store the path name of the referenced object. The path name for an object instance could be formed from the names of the properties that must be traversed to reach that object instance starting at the root. These path names would be similar to the path names in a file system. General object references would have to be made invalid if their target is moved or deleted.

8.5.17 Standard Streams

[TBS. This section will describe how standard streams, such as "SummaryInformation", are supported.]

8.5.18 Class Dictionary

[TBS. This section will describe how the AAFClassDictionary class and the related definition classes are mapped to structured storage. The definition classes are

[1] The notation "/Header/Content/Mobs" uses the IStorage and IStream naming convention described elsewhere in this document. This convention generates path names for stored objects and properties from property names. The property names are taken from the document "Proposed SMPTE Recommended Practice for Television – Interchange of Video and Audio Material and Related Descriptive Information as Edit Decision Data". This document contains the most up-to-date description of the AAF stored object model.

AAFDefinitionObject
    AAFClassDefinition
    AAFControlCodeDefinition
    AAFDataDefinition
    AAFEffectDefinition
    AAFPropertyDefinition
    AAFTypeDefinition]

8.5.19 Embedded Media

[TBS. This section will describe how embedded media is mapped to a structured storage file.]

8.5.20 Use Of Property Sets

[TBS. This design does not specify the use of property sets for representing stored objects. This section will describe the rationale for this design decision.]

8.5.21 AAF File SMPTE Signature

Some notes on the header of a structured storage file
1) The StructureStorageHeader is always in Intel byte order
2) The version number is 3.62 on NT, Macintosh and in the reference implementation of Structured Storage on Irix
3) StructuredStorageHeader._clid is null
4) A GUID may be written to StructuredStorageHeader._clid using fopen/fwrite
5) The presence of this GUID does not seem to affect StgIsStorageFile() or StgOpenStorage() et. al

All valid AAF files contain a SMPTE unique identifier in the StructuredStorageHeader._clid field.

The following GUID identifies a file as an AAF file It is
1) placed in the _clid field of the structured storage file header
2) 16 bytes in size
3) alway in Intel (little-endian) byte order (consistent with the rest of the structured storage file header) stored at byte offset 8 from the start of the file immediately follows the 8 byte structured storage file signature

```
// SMPTE identifier
// 06 0E 2B 34 01 01 01 04 20 46 41 41 2E 31 20 30
//
// AUID
// 20 46 41 41 2E 31 20 30 06 0E 2B 34 01 01 01 04
//
DEFINE_AUID(AAFFileSignature,
  0x20464141,
  0x2E31, 0x2030,
  0x06, 0x0E, 0x2B, 0x34, 0x01, 0x01, 0x01, 0x04)
```

This GUID appears as follows when the file is dumped in ASCII.

```
$ dump foo.aaf
       0    d0 cf 11 e0 a1 b1 1a e1 41 41 46 20 31 2e 30 20     ........AAF 1.0
      16    06 0e 2b 34 01 01 01 04 3e 00 03 00 fe ff 09 00     ..+4....>.......
...
$
```

8.6 Object Naming

[TBS. This section will describe the support provided by the *Object Manager* for naming objects. Object naming is used to support SMPTE Unique Labels (SULs) and SMPTE Unique Material Identifiers (UMIDs).]

8.7 Lazy Loading and Memory Reclamation

In this design lazy loading and memory reclamation are integrated with object references.

8.7.1 Lazy Loading

When an object is first created in memory from its persistent form on disk, only the primitive attributes are read.

Resolved object references are initialized to refer to the appropriate in-memory object. Unresolved object references are not initially followed in the hope that the object to which they refer will not be needed, hence the term “lazy loading”. Instead, unresolved object references are followed on demand. Following an unresolved object reference involves locating the referred to object in the object store, creating that object in memory and then initializing the reference to refer to the in-memory object. Lazy loading allows us to avoid doing this work if the referred to object is never needed. With respect to lazy loading, strong and weak object references are treated identically.

The *Object Manager* integrates lazy loading with object references. In this design an object reference may have the following states.

1. Void: no object is referenced. Void references are never followed.
2. Unresolved, object not loaded: refers to an object in the persistent store, that object does not exist in memory.
3. Unresolved, object loaded: refers to an object in the persistent store, that object exists in memory. When a reference in this state is followed it will be resolved to the in memory object. Note that when an object reference is in this state, the in-memory and persistent store instances of the referenced object may have different contents. This situation will occur if the in-memory object has been modified and has not yet been written out to persistent store. That is, when the in-memory instance of the referenced object is dirty.
4. Resolved, object not loaded: In this state the reference is stale and the object is reloaded.
5. Resolved, object loaded: in this state the referenced object is in memory and no work is needed when the reference is followed.

Note that every time a reference is followed the object directory is consulted to determine whether or not the object is loaded.

8.7.2 Memory Reclamation

Memory used by objects may be reclaimed once those objects have been written to persistent store. If this is performed via an object reference then the state of the reference will transition from “resolved” to “unresolved, object not loaded”. Note that this may cause other references to this object to transition from “resolved, object loaded” to “resolved, object not loaded”.

Memory reclamation may be performed when

- immediately complying with a request to load an object would exhaust free store
- free store is exhausted
- on demand
- free store utilization has reached a predetermined limit

8.8 Transient Objects

Transient objects are not associated with an object store. The state of a transient object is not saved across application invocations.

8.8.1 Rules for Combining Transient and Persistent Objects

1. Mixing objects from different object stores is not allowed.
2. A transient object may contain strong references only to other transient objects.

3. A transient object may contain weak references to either transient or persistent objects. Note that this means that a weak reference from a transient object will become invalid when the object store with which the referred to object is associated is closed.
4. A persistent object may contain strong references only to other persistent objects associated with the same object store.
5. A persistent object may contain weak references only to other persistent objects associated with the same object store.
6. Inserting a transient object into a persistent container object makes that object and recursively all objects that it references, persistent.

Transience is transitive over strong references. Persistence is transitive over both strong and weak references.

These rules are designed so that

1. Starting with a persistent object and recursively following all references, both strong and weak, one will encounter only persistent objects.
2. Every persistent object is referred to by at least one strong reference.

These conditions are necessary for transient objects to coexist with a correct implementation of recursive persistence.

8.8.2 How These Rules Are Implemented

An object belongs to the same file as the object that contains it. In that way objects from different files cannot be combined, and moving an object from one file to another can be accomplished by simple reattachment.

An object is transient if its containing object is transient, and persistent if its containing object is persistent. In that way persistent and transient object cannot be incorrectly combined, and changing the state (persistent vs. transient) of an object can be accomplished by simple reattachment.

8.9 Deleting Objects From an AAF File

[TBS. This section will describe how the *Object Manager* supports deleting objects from an AAF file.]

8.10 Copying Objects From One AAF File to Another

[TBS. This section will describe how the *Object Manager* supports copying objects from one AAF file to another.]

[Issues : Have to deal with 1) restricted weak references (including dictionary entries) 2) byte swapping if the source and target have different endianness.]

8.11 Moving Objects From One AAF File to Another

[TBS. This section will describe how the *Object Manager* Supports moving objects from one AAF file to another.]

8.12 COM Reference Counting

Since a design goal of the *Object Manager* is for transparency, the *Data Manager* code must treat the objects defined by the following two declarations identically…

```
ImplAAFComponent*                              _component1;
OMStrongReferenceProperty<ImplAAFComponent> _component2;
```

Since the COM rules require the *Data Manager* code to reference count `_component1` the transparency requirement means that the *Data Manager* code is also required to reference count `_component2`.

Transparency also requires that the *Data Manager* code must treat the objects defined by the following two declarations identically…

```
ImplAAFComponent* _components1[SIZE];
OMStrongReferenceVectorProperty<ImplAAFComponent> _components2;
```

Since the COM rules require the *Data Manager* code to reference count `_components1` the transparency requirement means that the *Data Manager* code is also required to reference count `_components2`.

Since correct reference counting mujst be implemented by the *Data Manager*, the *Object Manager* does no COM reference counting, in fact it designed indepently of whether it is managing COM or non-COM objects.

8.13 Object Directory

The object directory is a data structure that efficiently maps keys onto values. Possible implementations include a binary tree or a hash table. The key is an integer value, unique to a particular object directory that identifies a particular object in the table. Operations available on the object directory include

- store a value under a given key
- retrieve the value associated with a given key
- remove the value associated with a given key

The value stores the following information under each key

- the location in memory where the object was loaded
- the storage location from which the object was loaded
- validity information - “is the object loaded ?”, reference counts

The object directory is a per AAF file data structure. Note that there is not necessarily an entry in the object directory for each AAF object in a file. Entries are made in the directory as AAF objects are loaded. Entries are removed from the directory when they are no longer needed, for example, when an object is deleted.

Object references are implemented as keys that are valid in the appropriate object dictionary.

[TBS. More detail on the object directory will be provided here.]

8.14 Schema Evolution

[TBS. This section will describe how the *Object Manager* will support schema evolution - changes to the AAF object model and class hierarchy over time.]

8.15 Multiple Open AAF Files

[TBS. This section will describe how the *Object Manager* will support having more than one AAF file open at the same time.]

8.16 Shared Access to AAF Files

This section describes the support provided by the *Object Manager* for shared access to AAF files. It addresses such issues as the supported mixes of readers and writers. [TBS. Eventually this section will also address such issues as multi-thread and multi-process access and remote access.]

It is currrently possible to have an AAF file open multiple times for read only, with no writers. If an AAF file is open for write it may not be opened again either for read or for write. The longer-term intent is to also allow multiple readers with one writer.

If a reader is allowed to accesses a file that is being changed the issue of consistency arises. The AAF goal is to provide consistency at the object level. That is, the reader would see individual objects as they existed before or after a given change but not during a change.

Note that this would not mean that the reader would have a consistent view of the file. For example, the writer may delete an object, and all references to it, from the file but cannot do this atomically.

In order to provide object level consistency AAF file data (property values) and the AAF file meta-data (property indexes, vector and set indexes) need to be updated atomically.

While Microsoft's implementation of structured storage does provied a transacted mode on Microsoft platforms it does not do so on non-Microsoft platforms (Unix and Macintosh for example where the referenec implementation of structured storage is used). This means we need to "roll our own" object level transactions within AAF.

8.17 Class Dictionary

[TBS. This section will describe the design of the class dictionary component of the *Object Manager.*]

8.18 Object Lifetimes

[TBS. This section will describe the support provided by the *Object Manager* for managing object lifetimes through the use of reference counts.]

8.19 Media Streaming

[TBS. This section will describe *Object Manager* support for media streaming an will address issues such as structured storage stream alignment.]

8.20 Handling Failures

[TBS. This section will describe the detection and handling of errors within the *Object Manager.* It will also describe how those errors are reported to the clients of the *Object Manager.*]

8.20.1 Out of Disk Space

This error condition is of particular concern if it occurs while holding dirty objects.

[More TBS]

8.20.2 Out of Free Store

If this error condition occurs while holding dirty objects it must be possible to save them to disk without needing to consume additional free store.

[More TBS]

8.21 Testing

[TBS. We want to create a design that can be tested, describe how the implementation will be tested.]

8.22 Debugging

[TBS. We want to create a design that can be debugged, describe how the implementation will be debugged.]

8.23 Assertions

8.23.1 Overview Of Assertions

The AAF Object Manager makes extensive use of assertions. Monitoring of the assertions is enabled by the compilation symbol `OM_ENABLE_DEBUG`. The debug configuration of the AAF reference implementation defines this symbol.

The *Object Manager* bases its use of assertions on the concept of *design by contract* as described in *Object Oriented Software Construction 2nd Ed*. (by Bertrand Meyer). See Chapter 11 “Design by Contract: Building Reliable Software”.

8.23.1.1 Simple Assertions

[TBS. Describe simple assertions]

8.23.1.2 Routine Preconditions

[TBS. Describe routine preconditions]

8.23.1.3 Routine Postconditions

[TBS. Describe routine postconditions]

8.23.1.4 Routine Tracing

[TBS. Describe routine tracing]

8.23.2 Assertion Violation Backstop

8.23.2.1 Overview

The *Object Manager* behavior on an assertion violation is customizable (at compile time) but the default behavior on assertion violation is to throw an exception. However, such an exception cannot be allowed to propagate to the client code, so an assertion violation backstop is implemented.

The elements of the design are…

- An `AAF_INTERNAL_ERROR` code is defined
- An `OMAssertionViolation` class is defined
- The `OMAssertionViolation` is private to the *Object Manager*
- The Object Manager throws an instance of `OMAssertionViolation` when an assertion violation occurrs
- The `OMAssertionViolation` exception is not caught; instead it is allowed to hit the backstop.
- The backstop code is in the dodo generated com-api files (e.g. `CAAFHeader.cpp`). The backstop code catches the `OMAssertionViolation` exception and returns the `AAF_INTERNAL_ERROR` code.

8.23.2.2 Example Dodo Generated Code

```
class OMAssertionViolation;  // Opaque

HRESULT STDMETHODCALLTYPE
    CAAFHeader::LookupMob (aafUID_t *  pMobID, IAAFMob ** ppMob)
{
  HRESULT hr;
  ...
  ImplAAFMob * internalppMob = NULL;
  ImplAAFMob ** pinternalppMob = NULL;
  if (ppMob)
    {
      pinternalppMob = &internalppMob;
    }

  try {
    hr = ptr->LookupMob (pMobID, pinternalppMob);
  } catch (OMAssertionViolation& ) {
    hr = AAF_INTERNAL_ERROR;
  }
  ...
  return hr;
}
```

9. Notes for Developers of *Object Manager* Client Code

This section contains notes useful for developers writing *Object Manager* client code. In particular it addresses developers responsible for moving existing OMF code into the AAF *Data Manager*.

9.1 Cookbook for making Properties Persistent

This cookbook assumes you want to implement the (fictitious) class `AAFFoo` and already have the converted (from OMFI) code for the class. Also see `ImplAAFHeader.{cpp|h}` and `ImplAAFIdentification.{cpp|h}` for examples of classes where the properties have already been made persistent. You are also referred to section 6 of the "AAF *Object Manager* Design Specification".

Please note that, since some of the current *Object Manager* interfaces are still prototypes that

(a) This cookbook is not yet as simple as it should be.

(b) These instructions are subject to change - see tags of the form [CHANGE COMING:…].

9.1.1 Recipe (for the developer)

9.1.1.1 Use property declaration templates.

The following table illustrates the types to use.

SMPTE spec	Type	*Object Manager*
	Simple scalar	`OMFixedSizeProperty<>`
	Simple `struct`	`OMFixedSizeProperty<>`
	Unicode string (`wchar_t*`)	`OMWideStringProperty`
	String	`OMStringProperty`
	Array	`OMArrayProperty<>`
`StrongRef`	Strong reference	`OMStrongReferenceProperty<>`
`StrongRefArray`	Strong reference vector	`OMStrongReferenceVectorProperty<>`
`StrongRefSet`	Strong reference set	`OMStrongReferenceSetProperty<>`
`WeakRef`	Weak reference	`OMWeakReferenceProperty<>`
`WeakRefArray`	Weak reference vector	`OMWeakReferenceVectorProperty<>`
`WeakRefSet`	Weak reference set	`OMWeakReferenceSetProperty<>`

Note that `OMArrayProperty<>` is not yet implemented, please use `OMFixedSizeProperty<>` instead.
The *Object Manager* does not yet support the notion of sets and so `OMStrongReferenceSetProperty<>` and `OMWeakReferenceSetProperty<>` are not yet implemented. Instead please use `OMStrongReferenceVectorProperty<>` and `OMWeakReferenceVectorProperty<>.`
The *Object Manager* support for integrated weak references is not yet complete so please use `OMFixedSizeProperty<>` instead.
These templates are designed to provide access to persistent properties equivalent to non-persistent properties. That is, everything that may be done to `foo1` (see below) may also be done to `foo2` using exactly the same code. You should not have to explicitly call member functions on `OM*Property` classes.

```
Foo                       foo1;
OMFixedSizeProperty<Foo> foo2;
```

Assuming that class `AAFFoo` has properties `Apple` and `Pear`.

```
 OMFixedSizeProperty<Apple> _apple;
OMFixedSizeProperty<Pear>  _pear;
```

Note that in

```
OMStrongReferenceProperty<AAFoo> _f;
AAFFoo* _g;
```

The entity `_f` and the entity `_g` behave identically except that the object designated by `_f` is persistent whereas the one designated by `_g` is not.

Note also that the following are also equivalent except for persistence

```
OMStrongReferenceVectorProperty<AAFFoo> _v;
AAFoo* _w[SIZE];
```

9.1.1.2 Define property ids

Give each property a small integer (PID or "property id") and a name to identify it.
The integers must be unique within a property set instance. Since derived classes share the same property set as their base classes this means that the PIDs must be unique across a class and all of its base classes. If this rule is violated you'll get an assertion failure like the following...

```
Precondition "Property not already installed" failed in routine
"OMPropertySet::put".
```

You can do a run time check by running the COMModTestAAF application.
The names should be constructed as follows PID_<className>_<propertyname> (Please use the property names from the SMPTE spec as these reflect the most recent improvements to the AAF object model.)

```
const int PID_FOO_APPLE =  3;
const int PID_FOO_PEAR  = 14;
```

9.1.1.3 Initialize the Properties

Each property should be initialized with a property id and a name. Use the property names as given in the object specification.

```
ImplAAFFoo::ImplAAFFoo()
: _apple(PID_FOO_APPLE, "apple"),
  _pear( PID_FOO_PEAR,  "pear")
{
  ...
```

9.1.1.4 Initialize the Property Set (`_persistentProperties`)

```
ImplAAFFoo::ImplAAFFoo()
...
{
  _persistentProperties.put(_apple.address());
  _persistentProperties.put(_pear.address());
}
```

The property set is called **`_persistentProperties`** and is inherited by all AAF classes from `OMStorable` via **`ImplAAFObject.`**

9.1.2 Recipe (for the dodo tool)

9.1.2.1 Include the Appropriate Header Files

Dodo should have already done this for you. In your implementation header file, `ImplAAFFoo.h`, include `ImplAAFObject.h` and `OMProperty.h`

9.1.2.2 Declare the Class to be Storable

Dodo should have already done this for you. Invoke the macro `OMDECLARE_STORABLE()` [defined in `OMStorable.h`] in the public part of the `Impl` class declaration. Note that the macro invocation is **not** terminated with a semi-colon.

```
class ImplAAFFoo {
public:
  ...
  OMDECLARE_STORABLE(ImplAAFFoo)
```

```
  ...
private:
  ...
};
```

9.1.2.3 Define `OMStorable` Overrides

Dodo should have already done this for you. In the `ImplAAFFoo.cpp` file add the following...

```
extern "C" const aafClassID_t CLSID_AAFFoo;

OMDEFINE_STORABLE(ImplAAFFoo, CLSID_AAFFoo);
```

9.1.3 COM Reference Counting

9.1.3.1 An Example

The following is an example that defines a new type of segment that contains a single strong reference to a `Foo` object (an `ImplAAFFoo` pointer) and a vector of strong references to `Bar` objects (an array of `ImplAAFFoo` pointers). Both `ImplAAFFoo` and `ImplAAFBar` are subclasses of `ImplAAFObject`. The example illustrates the reference counting requirements on the *Data Manager* code. Please note that this example code has not been compiled.

The reference counting functions are `AcquireReference()`, `ReleaseReference()` and `ReferenceCount()` for all classes derived from either `ImplAAFRoot` or `AAFRoot`.

Observing the protocol shown in this example has the following benefits

1. Obeys all the COM reference counting rules
2. Prevents objects that are not in memory from being lazily loaded just so that they can be released.
3. Enables the *Object Manager* to check for and complain about strongly referenced objects that are deleted.

The reference counting rules shown in this example are the same rules that would apply if the following alternate declarations for `_foo` and `_bars` were used.

```
ImplAAFFoo* _foo;
ImplAAFBar* _bars[MAXSIZE];
```

9.1.3.1.1 Class Declaration

```
class ImplAAFFoo;
class ImplAAFBar;

class ImplAAFExample : public ImplAAFSegment
{
public:
  ImplAAFExample();

protected:
  ~ImplAAFExample();

public:
  void SetFoo(ImplAAFFoo* f);
  void GetFoo(ImplAAFFoo** f);
```

```
  void AppendBar(ImplAAFBar* f);
  void GetBarAt(ImplAAFBar** f, aafUInt32 n);
  ImplAAFBar* FindBar(bool (FindProc*)(ImplAAFBar* b) findProc);
  void AppendNewBar(void);

private:
  OMStrongReferenceProperty<ImplAAFFoo>       _foo;
  OMStrongReferenceVectorProperty<ImplAAFBar> _bars;

 };
```

9.1.3.1.2 Class Definition

```
ImplAAFExample:: ImplAAFExample() :
  _foo( PID_EXAMPLE_FOO, "foo"),
  _bars(PID_EXAMPLE_BARS,"bars")
{
}


ImplAAFExample::~ImplAAFExample()
{
  // Delete the contained Foo.
  ImplAAFFoo* oldFoo = _foo.setValue(0);
  if (oldFoo != 0)
    oldFoo->ReleaseReference();

  // Delete the contained array of Bars.
  size_t count = _bars.getSize();
  for (size_t i = 0; i < count; i++) {
    ImplAAFBar* oldBar = _bars.setValueAt(0, i);
    if (oldBar != 0)
      oldBar->ReleaseReference();
  }
}

void ImplAAFExample::SetFoo(ImplAAFFoo* f)
{
  ImplAAFFoo* oldFoo = _foo.setValue(f);
  if (oldFoo != 0)
    oldFoo->ReleaseReference();

  if (f != 0)
    f->AcquireReference();
}

void ImplAAFExample::GetFoo(ImplAAFFoo** f)
{
  *f = _foo;
  if (_foo != 0)
   _foo->AcquireReference();
}
```

```
void ImplAAFExample::AppendBar(ImplAAFBar* b)
{
  if (b != 0) {
    _bars.appendValue(b);
    b->AcquireReference();
  }
}

void ImplAAFExample::GetBarAt(ImplAAFFoo** f, aafUInt32 n)
{
  ImplAAFBar* t = 0;

  _bars.getValueAt(t, n);
  *f = t;

  if (t != 0)
    t->AcquireReference();
}

ImplAAFBar* ImplAAFExample::FindBar(
                                    bool (FindProc*)(ImplAAFBar* b) findProc)
{
  ImplAAFBar* b = 0;
  size_t count = _bars.getSize();
  for (size_t i = 0; i < count; i++) {
    _bars.getValueAt(b, i);
    if (b != 0) {
      if (findProc(b)) {
        b->AcquireReference();
        return b;
      }
    }
  }
  return 0;
}

void ImplAAFExample::AppendNewBar(void)
{
  // To be supplied
}
```

9.1.4 Notes

When compiling make sure your include path specifies `include/OM` and `src/OM`. This is a temporary measure the goal is to require that only include/OM be specified. The checked-in projects do this already. [CHANGE COMING: In future the OM source files will be organized such that you'll only have to specify include/OM.]

9.2 Changing Property Types

Here are some things to remember if you either

1. change the definition of a type when there are persistent properties of that type or
2. change the type of a persistent property

An example of 1 would be a change in the definition of aafTimeStamp_t (as was done recently). The _lastModified property of class AAFHeader is of type aafTimeStamp_t.

A hypothetical example of 2 would be to change the declaration of the lastModified property of class AAFHeader from

```
OMFixedSizeProperty<aafTimeStamp_t> _lastModified;
```

to

```
OMFixedSizeProperty<UGLY_SMPTE_HEXCodedTimeStamp_t> _lastModified;
```

The AAF Object Model, and as a consequence the *Object Manager* design and implementation, assumes that the type of a property, identified by a given property id, does not change over time. Or put another way, the *Object Manager* assumes that it can tell the type of a property data value stored in an AAF file from the associated stored property id.

So if an AAF file is created with a PID (property id) that corresponds to a given type, errors will occur if that file is read in by a toolkit compiled with a different definition of that type. Currently the error is detected only if the size of the type is changed. If the type is changed but the size remains the same, values in "old" files will be silently (and incorrectly) interpreted as values of the new type.

If you see the following error when you try to run either "ComAAFInfo" or "CppAAFInfo"

```
Assertion "Sizes match" failed in routine
"OMFixedSizeProperty<PropertyType>::restoreFrom".
The failure occurred at line 118 in file "../../../ref-
impl/src/OM\OMPropertyT.h".
The condition "size == _size" was violated.
```

You need to

1. run "ComClientTestAAF" to create a new Foo.aaf file
2. copy the newly created Foo.aaf file from AAFWinSDK/examples/com-api/ComClientTestAAF to AAFWinSDK/examples/com-api/ComAAFInfo
3. run "ComAAFInfo" again - this time you shouldn't see the error message
4. repeat 1 - 3 for "CppClientTestAAF" and "CppAAFInfo"

This is currently only a development issue i.e. it occurs during development while the API is still evolving. We will face a similar issue when we want to release a version 2.0 toolkit after users have created files with a version 1.0 toolkit.

During development I suggest we handle this problem by assigning a new PID to any properties whose type we change. I'll add code to the OM so that, given a changed PID, the error will always be detected.

9.3 Persistent Objects, Attached Objects And Files

Sometimes in *Data Manager* code you may want to determine if a particular object is associated with an on disk file (is persistent) , is attached to (owned by) another object or is contained within a file.

9.3.1 Determining if an Object is Owned by Another Object

An example of the need to determine if an object is owned by another object occurs in `ImplAAFSequence::AppendComponent()`. This note also applies to the implementation of other append methods in the *Data Manager*. Since an object may have only one owner all append functions should check to see that the object they are being passed is not already owned. The function "`bool OMStorable::attached()`" is used for this purpose. This function returns `true` if '`this`' is an attached object, otherwise it returns `false`. Here's what the *Data Manager* code should look like

```
AAFRESULT STDMETHODCALLTYPE
    ImplAAFSequence::AppendComponent (ImplAAFComponent* pComponent)
{
  ...
   if (pComponent->attached())
     return AAFRESULT_OBJECT_ALREADY_ATTACHED;
  ...
}
```

If this check is omitted from the *Data Manager* code, the actual attempt to attach the object will fail with an assertion violation. The assertion violation only occurs when assertions are enabled. Assertions are enabled in a debug build, they are disabled in a release build.
Usually the *Data Manager* code wants to check that an object is not already attached but sometimes it may require that an object be attached in this case the code is as follows.

```
if (!pComponent->attached())
     return AAFRESULT_OBJECT_NOT_ATTACHED;
```

9.3.2 Determining If An Object Is Contained Within A File

To determine if an object is within a file the Data manager should use code like the following…

```
if (!pObj->inFile())
     return AAFRESULT_OBJECT_NOT_IN_FILE;
```

9.3.3 Determining if an Object is Persistent

An example of the need to determine if an object is persistent occurs in the implementation of media access. Since AAF does not support transient media, the media access code should check that the object on which media access is being attempted is in fact persistent. The function "`bool OMStorable::persistent()`" is used for this purpose. This function returns `true` if '`this`' is a persistent object, otherwise it returns `false`. Here's an actual example from the *Data Manager* code.

```
 AAFRESULT STDMETHODCALLTYPE
   ImplAAFEssenceData::Read (aafUInt32  bytes,
                             aafDataBuffer_t  buffer,
                             aafUInt32 *bytesRead)
{
  ...
  // Cannot access the data property if it is NOT associated with a file.
  if (!persistent())
    return AAFRESULT_OBJECT_NOT_PERSISTENT;
  ...
}
```

If this check is omitted from the *Data Manager* code subsequent calls to the *Object Manager* to access media on non-persistent (i.e. transient) objects will fail with an assertion violation. The assertion violation only occurs when assertions are enabled. Assertions are enabled in a debug build, they are disabled in a release build.

9.3.4 Summary

OMStorable Function	Precondition	Error Code
attached()	None	`OBJECT_NOT_ATTACHED`
inFile()	attached()	`OBJECT_NOT_IN_FILE`

persistent()	inFile()	OBJECT_NOT_PERSISTENT

The error codes listed are those for which a false result from the OMStorable functions denotes an error. The code listed omits the AAFRESULT_ prefix.

9.3.5 Notes

1. For an object to be persistent it is necessarily in a file.
2. For an object to be in a file it is necessarily attached.
3. An object may be attached but not persistent.
4. An object may be attached but not in a file.

10. Performance, Capacity and Scalability Tests

10.1 Object capacity

10.1.1 purpose of test

Determine if there are any built-in capacity limits on the number of objects that can be created, held in memory and/or stored in a file.

10.1.2 ideal behavior

No limits.

10.1.3 expected behavior

The structured storage limit of approximately 2k open objects.

10.1.4 planned optimization

Keep in memory objects closed, even if they are dirty, open each object before saving it and close it afterwards.

10.1.5 program

Creates a named file and creates a given number of objects and saves them in the file.

10.1.6 input data

None - created on the fly.

10.1.7 graph

None.

10.2 File open latency

10.2.1 purpose of test

Investigate how the time to open a file varies as the number of objects in the file increases.

10.2.2 ideal behavior

The file open time is independent O(1)] of the number of objects in the file.

10.2.3 expected behavior

Ideal.

10.2.4 planned optimization

None.

10.2.5 program

Opens a named file, times the open, then calls close.

10.2.6 input data

Several differently sized files each containing a known number of objects. The files should all have similar structure.

10.2.7 graph

X = number of objects, Y = file open time

10.3 File save latency (create)

10.3.1 purpose of test

Investigate how the time to save a file varies as the number of objects in the file increases.

10.3.2 ideal behavior

The file save time is a linear function [O(n)] of the total number of objects in the file.

10.3.3 expected behavior

Worse than ideal (some function of the total number of properties in the file).

10.3.4 planned optimization

Write whole objects instead of whole properties.

10.3.5 program

Creates a named file containing a given number of objects, saves the file and times the save, then calls close.

10.3.6 input data

None - created on the fly.

10.3.7 graph

a) X = number of objects, Y = file save time, and
b) X = number of propeties, Y = file save time

10.4 File save latency (modify)

10.4.1 purpose of test

Investigate how the time to save a file varies as the number of dirty objects in the file increases.

10.4.2 ideal behavior

The file save time is a linear function [O(n)] of the number of dirty objects in the file.

10.4.3 expected behavior

The file save time is a linear function [O(n)] of the total number of objects, clean and dirty, in the file.

10.4.4 planned optimization

Implement a dirty bit, write only dirty objects.

10.4.5 program

Creates a named file containing a given number of objects, saves the file (not timed). Next dirties a known number of objects (say 25% of the total) by changing a property, calls save, times the save, then calls close.

10.4.6 input data

None - created on the fly.

10.4.7 graph

a) X = number of objects, Y = file save time and
b) X = number of dirty objects, Y = file save time

10.5 Vector/set scalability

10.5.1 purpose of test

Investigate how the following operations on vectors/sets vary as the number of objects in the vector/set increases.

- add a new object to the vector/set
- remove a given object from the vector/set
- find a given object in the vector/set

· create large vector/set (many add operations)

10.5.2 ideal behavior

· TBS this should be a table with colums for vector and set
· add O(lg n)
· remove O(lg n)
· find O(lg n)
· create large O(n lg n)

10.5.3 expected behavior

For some operations - O(n) since linear searches are currently employed. Worse [O(n^2)] for "create large" where growing currently includes copying of elements that are already present.

10.5.4 planned optimization

Mostly balanced binary tree (red-black tree) implementation of vectors/sets giving nearly ideal behavior. TBS can't use tree for vectors that aren't sparse.

10.5.5 program

Create a large vector containing a specified number of object, measure this creation time. Time the add, remove and find, operations. Repeat for other vector sizes.

10.5.6 input data

None - created on the fly.

10.5.7 graph

a) For op = (add, remove, find) X = number of objects in the vector/set, Y = time to perform op, and
b) X = number of objects, Y = time to create vector/set containing that number of objects.

10.6 Essence access (write)

10.6.1 purpose of test

Determine the rate at which essence data can be written to a file (bytes/second).

10.6.2 ideal behavior

Meets AAF requirement of ? bytes/second.

10.6.3 expected behavior

Too slow, since the implementation of structured storage currently in use doesn't support "unbuffered I/O".

10.6.4 planned optimization

Possibly use the "4k sector size" implementation of structured storage, however this is not backwardly compatible with the current implementation of structured storage.

10.6.5 program

Generates plausible, but fake, essence data in memory and writes it to an aaf file measuring the output rate.

10.6.6 input data

None - created on the fly.

10.6.7 graph

None.

10.7 Essence access (write)

10.7.1 purpose of test

Determine the rate at which essence data can be read from a file (bytes/second).

10.7.2 ideal behavior

Meets AAF requirement of ? bytes/second.

10.7.3 expected behavior

Too slow, since the implementation of structured storage currently in use doesn't support "unbuffered I/O".

10.7.4 planned optimization

Possibly use the "4k sector size" implementation of structured storage, however this is not backwardly compatible with the current implementation of structured storage.

10.7.5 program

Reads essence data from a file and measures the input rate.

10.7.6 input data

Use the file(s) created by the essence access (write) test above.

10.7.7 graph

None.

11. Implementation Order

This section proposes an implementation order for the *Object Manager* functionality. The goal is to choose an order that results in the shortest time to create an implementation that can read and write AAF files (not necessarily in their final format).

1) Stubs only implementation
   - Call all interfaces but no functionality
   - Works on all supported platforms
2) File open/create and close
   - Create a file or open an existing AAF file
   - Cannot write or read objects
3) Write persistence - simple properties
   - Create an instance of any registered AAF class and save (simple properties only) it to an AAF file
4) Read persistence - simple properties
   - Read (simple properties only) an instance of any registered AAF class from a previously created AAF file
   - Cannot modify the object
5) Read/write persistence - simple properties
   - Read and write (simple properties only) an instance of any registered AAF class to and from an AAF file
6) Read/write persistence - strong references
   - Read and write an instance of any registered AAF class (including strong references) to and from an AAF file
7) Read/write persistence - strong reference vectors
   - Read and write an instance of any registered AAF class (including strong reference vectors) to and from an AAF file
8) Read/write persistence - weak references (restricted)
   - Read and write an instance of any registered AAF class (including restricted weak references) to and from an AAF file
9) Read/write persistence - media data
   - Read and write an instance of any registered AAF class (including media data) to and from an AAF file
10) Read/modify/write (write all objects)
   - Read an object from a file, modify a property, write all objects
11) Read/modify/write (write only changed objects)
   - Read an object from a file, modify a property, write only the changed object
12) Read/modify/write (write only changed properties)
   - Read an object from a file, modify a property, write only the changed property
13) Single file persistence
   - Copy objects
   - Move objects

14) Lazy loading at the object level
    - Load only those objects that are accessed
15) Lazy loading at the property level
    - Load only those properties that are accessed
16) Transient objects
    - Create transient objects (objects not associated with any file and that won't be persisted)
    - Create a transient object and then insert it into a persistent collection object and have the object persisted
    - Remove an object from a persistent collection object and have it become non-persistent (transient)
17) More than one file open at a time
    - Have more than one file open at the same time
    - Objects are persisted to and from the proper file
    - Cannot copy of move objects from one file to another
18) Multi-file persistence - copy/move objects between files
    - File to file object copy
    - File to file object move
19) Persistence of objects to which optional properties have been added
    - Define new properties for an existing class
    - Write and read instances of that class
20) Persistence of instances of user defined classes - with no user defined behavior
    - Define a new class
    - Write and read instances of that class
21) Persistence of instances of user defined classes - with user defined behavior (extended classes)
    - Define a new class, derived from a predefined AAF class, override a virtual function
    - Have the user defined virtual function called by the AAF tool kit
22) Reading of instances of user defined classes without the creation code
    - Write an instance of a user defined class from one application
    - Read that instance in another application that does not have the object creation code
23) Memory reclamation (lazy unloading)
    - [TBS.]
24) Storage optimizations
    - [TBS.]
25) Garbage collection
    - [TBS.]
26) Weak references (general)
    - [TBS.]
27) Schema evolution
    - [TBS.]

12. Glossary

AAF: Advanced Authoring Format.

AAF Class Dictionary: Same as AAF Dictionary.

AAF Dictionary: A data structure describing all AAF classes and their properties. Both predefined and user defined classes are described in the AAF Dictionary.

Ancestor: [Definition TBS.]

API: Strictly - Applications Programming Interface, more loosely - Programming Interface.

Class Dictionary: Same as AAF Dictionary.

COM: Component Object Model.

Container: Either container file on disk or container object. In this design - container object.

Descendant: [Definition TBS.]

Free store: Dynamically allocated memory, also called the heap.

Isomorphic persistence: An approach to object persistence in which the shape of the graph defined by the objects and their object references is preserved. In implementation language terms, isomorphic persistence preserves pointer identity.

IStorage: [Definition TBS.]

IStream: [Definition TBS.]

Object reference: The implementation of an association between objects. An object reference has both an in-memory and an on-disk form. There are two kinds of object reference, strong and weak.

Object store: The place to which persistent objects are saved. A disk file. In the context of this document the term *object store* is synonymous with *AAF file*.

Persistent object: Persistent objects are objects that are associated with an object store. The state of a persistent object is saved across application invocations. Also known as *linked* objects.

Persistent store: Same as object store.

Primitive Type: For the purposes of this specification, any type not descended directly or indirectly from AAFObject. Primitive types are the building blocks used to create other types.

Property: [Definition TBS.]

Recursive persistence: An approach to object persistence in which all of the objects associated with a given object store may be found by starting at a root object and recursively following all references, both strong and weak.

Semantically valid: [Definition TBS.]

Stable times: Those times at which an object may be observed by other objects. The term "stable times" means between, and not during updates. As an example, in a doubly linked list, the condition (next->previous == this), which is one of the invariants that defines a doubly linked list, holds only at stable times. The condition does not hold during the removal of an element.

Strong object reference: An object reference that implements the “contains” association. Strong object references connote ownership. Since an object may have only one owner there may be, at most, only one strong reference to a given object. Compare with *weak object reference*.

Structurally valid: [Definition TBS.]

TBS: To Be Supplied, To Be Specified.

Transient object: Transient objects are objects that are not associated with an object store. The state of a transient object is not saved across application invocations. Also known as *unlinked* objects.

Weak object reference: An object reference that implements an association between objects. Weak object references do not connote ownership. There may be zero or more weak reference to a given object. Compare with *strong object reference*.

13. References

13.1 General References

· AAF Web site - http://www.AAFAssociation.org/
· OMF Web site - http://www.omfi.org

13.2 COM and Structured Storage

· “Inside Distributed COM”, Guy Eddon, Henry Eddon, 1998 Microsoft Press, ISBN1-57231-849-X

See pages 277-286 for an overview of structured storage. See also chapter 7 – “monikers and structured storage”.

· "Inside OLE, 2nd Ed.", Kraig Brockschmidt, 1995, Microsoft Press, ISBN 1-55615-843-2
See pages 35-38 for a very high level overview of structured storage. See the whole of chapter 7 for a detailed look at structured storage.

· "Understanding ActiveX and OLE", David Chappell, 1996, Microsoft Press, ISBN 1-57231-216-5
See chapter 5 "Persistence".

· "Essential COM", Don Box, 1998, Addison Wesley, ISBN 0-201-63446-5
See Chapter 2 "Interfaces" and in particular the section entitled "Resource Management and IUnknown" for a clear description of the COM reference counting rules.

13.3 Object Oriented Software Engineering

· "Object Oriented Software Construction, 2nd Ed.", Bertrand Meyer, 1997, Prentice Hall, ISBN 0-13-629155-4
See chapter 31 "Object Persistence and Databases".

13.4 Object Databases

· "The Object Database Standard: ODMG 2.0", R. G. G. Cattell and Douglas K. Barry (eds), 1997, Morgan Kaufmann, ISBN 1-55860-463-4

13.5 Design Patterns

· "Design Patterns : Elements of Reusable Object Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, 1994, Addison Wesley, ISBN 0-201-63361-2

13.6 Program Portability, Data Representation And Data Exchange

· "C – A Reference Manual", Samuel P. Harbison and Guy L. Steele Jr., 1991, Prentice Hall, ISBN 0-13-110933-2
See chapter 6 "Conversions and Representations". In particular see section 6.1.2 "Byte Ordering" and section 6.1.3 "Alignment Restrictions". See also chapter 5 "Types".

· "A Retargetable C Compiler : Design and Implementation", Christopher Fraser and David Hanson, 1995, Benjamin/Cummings, ISBN 0-8053-1670-1
See chapter 11 "Declarations". In particular see section 11.5 "Structure Specifiers".

· "See MIPS Run", Dominic Sweetman, 1999) Morgan Kaufmann, ISBN 1558604103

13.7 Data Structures

· "Introduction to Algorithms", Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, 1997 MIT Press (McGraw-Hill). ISBN 0-262-03141-8
See chapter 14, page 263 for an excellent presentation of Red Black Trees.

· "The Modula-2 Software Component Library, Volume 3", Charles Lins, 1989 Springer-Verlag, ISBN 0-387-97074-6
· "Algorithms + Data Structures = Programs", Niklaus Wirth, 1976, Prentice Hall, ISBN 0-13-022418-9
· "Algorithms and Data Structures", Nlklaus Wirth, 1986 Prentice Hall, ISBN 0-13-022005-1
· "The Design and Analysis of Computer Algorithms", Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, 1974 Addison Wesley, ISBN 0-201000029-6

Related Documents

Document Name	Location	Owner	Description
Proposed SMPTE Recommended Practice for Television – Interchange of Video and Audio Material and Related Descriptive Information as Edit Decision Data		Josh Goldman	A proposed SMPTE standard. Describes the stored representation of the AAF object model. This description is more up-to-date than the "AAF Object Specification".
AAF Object Specification	AAF Web site	Josh Goldman	Describes the stored representation of the AAF object model.
Microsoft Advanced Authoring File Format Requirements Document		Microsoft	
Microsoft Multimedia Task Force AAF Requirements Addendum 1		Microsoft	
AAF Object Management		Oliver Morgan	
AAF Toolkit Architecture		Bob Tillman	
AAF Plug In Issues		Oliver Morgan	
OMFI/Structured Storage Analysis		Microsoft	
OMF Interchange Specification Version 2.1		Josh Goldman	

14. Revision History

Name	Date	Version	Description
Tim Bingham	5/26/98	0	Outline only
Tim Bingham	5/28/98	0.1	Start adding content (not published)
Tim Bingham	5/29/98	0.2	Add more content (not published)
Tim Bingham	6/1/98	0.3	Get ready for preliminary review (not published)
Tim Bingham	6/2/98	0.4	Preliminary review version (depth charge)
Tim Bingham	6/23/98	0.5	Incorporate review comments
Tim Bingham	7/13/98	0.6	Incorporate more review comments (not published)
Tim Bingham	7/27/98	0.7	Provide details of OM interfaces, more merging.
Tim Bingham		0.8	Add more detail on design of OM interfaces
Tim Bingham	3/10/99	0.9	Add and expand cookbook. Add more detail on property types. Add information on storage overhead. Add more detail to “Class Interfaces” section. Add more detail, including data structures, on mapping of AAF objects to structured storage. Add more references. Update and expand the summary of requirements. Add information on mapping between SMPTE unique identifiers and AUIDs. Record the resolution of all open issues.
Tim Bingham	3/10/99	1.0	Update and expand the summary of requirements. Remove review comments section, which was previously included for historical reasons. Add section on AAF API file save semantics. Remove section on possible implementations of AAFDictionary::createInstance(). Update some of the code fragments. Add a reiteration of the rules for file byte order. Add missing requirements on byte ordering, “foreign objects”, embedding and *Object Manager* interfaces. Update section on semantics of AAFFile::Save(). Many small changes to improve consistency. More cookbook improvements. Remove “dependencies on other AAF components” section since this material is now all covered elsewhere in the document. General editorial clean up pass.
Tim Bingham	7/9/99	1.1	Move “Object Creation” to “Class Interfaces” section. Add description of OMType to “class interfaces” section. Add information on COM reference counting to the design section. Added “Reference Counting Cookbook” to “Developer Notes” section. Add section on the design of optional property support. Expand references section. Add description of the “assertion violation backstop” design. Use “type” for “data type” (of a property value) and “stored form” for the type of on-disk representation used for the property value.
Tim Bingham	8/24/99	1.2	Added new section “Creating Objects and Meta Data Objects”.
Tim Bingham	9/22/99	1.3	Add new section on “Indirect, private, encrypted, opaque and KLV types”. Fill out section on “Shared Access to AAF Files”. Added new section on “AAF File SMPTE Signature”. Added new section on “File Mode Flags”. Add section on “Performance, Capacity and Scalability Tests”.
Tim Bingham	11/3/99	1.4	Add design information on strong reference sets and on weak references.
Tim Bingham	4/5/00	1.5	Update description of stored property set and collection (strong and weak reference vectors and sets) indexes.
Tim Bingham	6/4/01	1.6	Prepare for release to Open Source.