Object Manager Stored Format

Avid Technology Inc.

Revision 0.4

Author: Tim Bingham

August 14, 2001

Contents

1. Mapping of Objects to Structured Storage

1.1 Overview

1) Each object is represented by a corresponding IStorage. The stored id of the object is stored as the CLSID of the IStorage object and is part of the structured storage overhead.
2) Each IStorage contains an IStream called "properties". The “properties” IStream is consists of two parts, the first portion contains the index of properties for the object and the second portion contains the “flat” or “simple” property values for the object. “Flat” and “simple” here means values that are not objects, that are not object collections and that are not streams. Note, however, that objects, object collections and streams do contribute to the “properties” IStream. The index and values are in the same IStream, rather than in separate IStreams, to reduce the Structured Storage overhead.
    a) The property index portion contains a header followed by a counted array of structures.
        i) The header has the format.
            (1) Byte order
            (2) Count of properties. The number of array elements that follow.
        ii) The counted array has the format with the following fields
            (1) Property Id – identifies the property
            (2) Property stored form - the structural “type” of the property. This indicates the meaning of the “flat” value in the “properties” stream.
            (3) Size – the size of the “flat” value of this property in the “properies” IStream.
    b) The property value portion contains the “flat values” of the properties for this object.
3) A single contained object is stored in a sub-IStorage. The name of the IStorage is given by the “flat” value in the “properties” IStream corresponding to the contained object’s entry in the "properties" IStream.
4) A contained vector of objects is represented as follows
    a) Each vector is described by an index stored in an IStream. The name of the vector index IStream is given by the “flat” value in the “properties” IStream corresponding to the contained object vector’s entry in the "properties" IStream.
    b) The contents of the vector index IStream are
        i) Count of objects
        ii) First free insertion key
        iii) Last free insertion key
        iv) Array of insertion key values, one for each contained object, the first key in the array is the key of the first object in the contained vector and so on.
    c) Each sub-object in the contained vector is stored in an IStorage whose name is formed from the name of the vector and the insertion key of the contained object.
5) A contained sets of objects is represented as follows.
    a) Each set is described by an index stored in an IStream. The name of the set index IStream is given by the “flat” value in the “properties” IStream corresponding to the contained object set’s entry in the "properties" IStream.
    b) The contents of the set index IStream are
        i) Count of objects
        ii) First free insertion key
        iii) Last free insertion key
        iv) Key property id – the id of the property used to uniquely identify each object in the set. The value of this property is the object’s search key
        v) Key size – the size of the search key

vi) Array of triples, one for each contained object
    (1) Insertion key
    (2) Count of weak references
    (3) Key value – the search key

6) Weak references are represented as follows
    a) Tag – identifies the path from the root object to the property instance containing the object that is the target of this weak reference
    b) Key property id
    c) Key size
    d) Key value
7) Vectors and sets of weak references are represented as follows.
    a) Count of referenced objects
    b) Tag – identifies the path from the root object to the property instance containing the object that is the target of this weak reference
    c) Key property id
    d) Key size
    e) Array of key values one for each referenced object
8) Media data is stored in a sub-IStream. The name of the IStream is given by the “flat” value in the “properties” IStream corresponding to the Media data stream’s entry in the "properties" IStream
9) There is one per-file data structure used for resolving weak references. This data structure is stored in an IStream called “referenced properties” in the root IStorage. This stream consists of
    a) Byte order
    b) Count of entries
    c) A sequence of null terminated lists of property ids. The first list is referenced using tag 0 and so on. Each list is a path from the root object to a particular property instance.

1.2 Data Structures

This section describes the data structures used to map objects on to structure storage. Note that the `typdefs` shown here are not real type definitions from any implementation, they are provided for illustrative purposes only.

1.2.1.1 Integral Types

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

```
typedef …         OMUInt8;
typedef …         OMUInt16;
typedef …         OMUInt32;
```

1.2.1.2 Data Types

These types are used to define members of data structures.

```
typedef OMUInt8  OMByteOrder;
typedef OMUInt8  OMVersion;
typedef OMUInt16 OMPropertyCount;
typedef OMUInt16 OMPropertyId;
typedef OMUInt16 OMStoredForm;
typedef OMUInt16 OMPropertySize;
typedef OMUInt8  OMKeySize;
typedef OMUInt16 OMPropertyTag;
typedef OMUInt16 OMCharacter;
```

1.2.1.3 Property Index

1.2.1.3.1 Purpose

The property index is an index into the property values. Both the index and the values (“flat” values only) are stored in a stream named “properties”.

1.2.1.3.2 External representation

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

1.2.1.3.3 Structure of Property Index Header

A `PropertyIndexHeader` is defined as follows…

```
typedef struct PropertyIndexHeader {
  OMByteOrder     _byteOrder;     // 1 byte
  OMVersion       _formatVersion; // 1 byte
  OMPropertyCount _entryCount;    // 2 bytes
} 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.

1.2.1.3.4 Structure of a Property Index Entry

```
typedef struct PropertyIndexEntry {
  OMPropertyId   _pid;           // 2 bytes
  OMStoredForm   _storedForm;    // 2 bytes
  OMPropertySize _length;        // 2 bytes
} PropertyIndexEntry;
```

The `_pid` is the id that describes the property. This is a shorthand version of the GUID that uniquely identifies the property. Property ids are locally unique. For all predefined properties the property id is the same in all 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 "flat" 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 dictionary.
The `_length` is the length, in bytes, of the property value in the property value stream.

1.2.1.4 Strong Object Reference

1.2.1.4.1 Purpose

A single contained object.

1.2.1.4.2 External Representation

Stored form	SF_STRONG_OBJECT_REFERENCE
Property value	Name of object

1.2.1.5 Strong Object Reference Vector

1.2.1.5.1 Purpose

An ordered collection of strongly referenced (contained) objects.

1.2.1.5.2 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.

1.2.1.5.3 Structure of a Strong Object Reference Vector Index Header

A `StrongReferenceVectorIndexHeader` is defined as follows…

```
typedef struct StrongReferenceVectorIndexHeader {
  OMUInt32 _entryCount;          // 4 bytes
  OMUInt32 _firstFreeKey;        // 4 bytes
  OMUInt32 _lastFreeKey;         // 4 bytes
} StrongReferenceVectorIndexHeader;
```

The `_entryCount` is the number of `VectorIndexEntry` structs that follow.
The `_firstFreeKey` is the next local key that will be assigned in this vector.
The `_lastFreeKey` is the highest unassigned key above `_firstFreeKey`. The keys between `_firstFreeKey` and `_lastFreeKey` are unassigned, while there may be other gaps in key assignement this represents the largest one.

1.2.1.5.4 Structure of a Strong Object Reference Vector Index Entry

```
typedef struct StrongReferenceVectorIndexEntry {
  OMUInt32 _localKey;             // 4 bytes
} 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 corresponding 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.

1.2.1.6 Strong Object Reference Sets

1.2.1.6.1 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

1.2.1.6.2 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>}

Each set index consists of a `StrongReferenceSetIndexHeader` followed by `_entryCount` `StronReferenceSetIndexEntry` structs.

1.2.1.6.3 Structure of a Strong Object Reference Set Index Header

```
typedef struct StrongReferenceSetIndexHeader {
  OMUInt32     _entryCount;         // 4 bytes
  OMUInt32     _firstFreeKey;       // 4 bytes
  OMUInt32     _lastFreeKey;        // 4 bytes
  OMPropertyId _identificationPid;  // 2 bytes
  OMKeySize    _identificationSize; // 1 byte
} 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.

1.2.1.6.4 Structure of a Strong Object Reference Set Index Entry

```
typedef struct StrongReferenceSetIndexEntry {
  OMUInt32     _localKey;           // 4 bytes
  OMUInt32     _referenceCount;     // 4 bytes
  <variable>   _identification;     // N bytes
} 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.

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.

1.2.1.7 Weak Object Reference

1.2.1.7.1 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.

1.2.1.7.2 External representation

Stored form	SF_WEAK_OBJECT_REFERENCE

1.2.1.7.3 Structure of a Weak Object Reference

```
typedef struct WeakObjectReference {
  OMPropertyTag _referencedPropertyIndex;  // 2 bytes
  OMPropertyId  _identificationPid;        // 2 bytes
  OMKeySize     _identificationSize;       // 1 byte
  <variable>    _identification;           // N bytes
} WeakObjectReference;
```

The `_referencedPropertyIndex` is the index into the referenced property table of the path to 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.

1.2.1.8 Weak Object Reference Vector

1.2.1.8.1 Purpose

An ordered collection of weak references.

1.2.1.8.2 External representation

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

1.2.1.8.3 Structure of a Weak Reference Vector Index Header

```
typedef struct WeakReferenceVectorIndexHeader {
  OMUInt32      _entryCount;              // 4 bytes
  OMPropertyTag _referencedPropertyIndex; // 2 bytes
  OMPropertyId  _identificationPid;       // 2 bytes
  OMKeySize     _identificationSize;      // 1 byte
} WeakReferenceVectorIndexHeader;
```

1.2.1.8.4 Structure of a Weak Object Reference Vector Index Entry

```
typedef struct WeakReferenceVectorIndexEntry {
  <variable> _identification;             // N bytes
} WeakReferenceVectorIndexEntry;
```

1.2.1.9 Weak Object Reference Set

1.2.1.9.1 Purpose

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

- efficiently located by key - O(lg N)

1.2.1.9.2 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

1.2.1.9.3 Structure of a Weak Object Reference Set Index Header

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

1.2.1.9.4 Structure of a Weak Object Reference Set Index Entry

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

1.2.1.10 Data Stream (Media Data)

1.2.1.10.1 Purpose

Storing embedded media. Also used to store other large variably sized information such as timecode.

1.2.1.10.2 External Representation

Stored form	SF_DATA_STREAM
Property value	Name of stream

1.2.1.10.3 Structure of a Data Stream

```
typedef struct DataStream {
  OMByteOrder             _byteOrder;  // 1 byte
  OMCharacter[<variable>] _streamName; // N bytes
} DataStream;
```

The `_streamName` is expressed as a null terminated string of 2-byte UNICODE characters. The size of the `_streamName` is given by 2 * (number of characters + 1). The maximum size is 32 bytes, this is a Structured Storage constraint. Note that the `_byteOrder` is the byte order of the data in the stream. The byte order of the `_streamName` is given by the `_byteOrder` field of the `PropertyIndexHeader`.

1.2.1.11 The Referenced-Properties Table

A weak object reference references an object in a particular strong reference set property instance. Property instances are represented by a null terminated list of property ids. The list is the path from the root object to the property instance. In order to avoid storing the path to the referenced property in each weak reference the path is stored once, in the referenced-properties table, and the index of the path in the table is stored in the weak reference. This index is also called a tag.

There is one referenced-properties table in each AAF file. The referenced-properties table is a stream called "`/referenced properties`". The stream consists of a header followed by a sequence of null terminated property id lists similar to a string space.

1.2.1.12 The Referenced-Properties Table Header

```
typedef struct ReferencedPropertiesTableHeader {
  OMByteOrder     _byteOrder;      // 1 byte
  OMPropertyCount _pathCount;      // 2 bytes
  OMUInt32        _pidCount;       // 4 bytes
} ReferencedPropertiesTableHeader;
```

The `_pathCount` field holds the number of referenced-properties in the table. Each reference property is stored as a property path – a null-terminated list of property ids. The `_pidCount` field is the total number of property ids that follow, including null terminators. The first property path in the list has a referenced property index (tag) of 0 and so on.

1.3 Storage and Stream Naming

[TBS]

1.4 Stored Forms

This section describes the stored form bit assignments.

1.4.1 Assignments

bit(s)	Value
15..8	Not used, must be 0
7..6	00 = object reference
	01 = stream
	10 = fixed size data
	11 = variable size data
5	0 = weak
	1 = strong
4	0 = singleton
	1 = collection
3	0 = vector
	1 = set
2	0 = not a unique object identification (set or search key)
	1 = a unique object identification (set or search key)
1	0 = opaque (not understood or interpreted by the Object Manager)
	1 = transparent (understood and interpreted by the Object Manager)
0	0 = not a stored object identification ("CLSID")
	1 = a stored object identification ("CLSID")

1.4.1.1 Notes

1) Not all combinations are valid
   a) bit 5 is only examined if bits 7..6 == 00
   b) bit 3 is only examined if bit 4 == 1
2) Not all valid combinations are currently used/implemented

1.4.2 Currently Defined Values

Stored form name	Value	Value	Used
SF_DATA	10.x.x.x.x.1.0	82	y
SF_DATA_STREAM	01.x.x.x.x.1.0	42	y
SF_STRONG_OBJECT_REFERENCE	00.1.0.x.x.1.0	22	y
SF_STRONG_OBJECT_REFERENCE_VECTOR	00.1.1.0.x.1.0	32	y
SF_STRONG_OBJECT_REFERENCE_SET	00.1.1.1.x.1.0	3A	y
SF_WEAK_OBJECT_REFERENCE	00.0.0.x.x.1.0	02	y
SF_WEAK_OBJECT_REFERENCE_VECTOR	00.0.1.0.x.1.0	12	y
SF_WEAK_OBJECT_REFERENCE_SET	00.0.1.1.x.1.0	1A	y
SF_WEAK_OBJECT_REFERENCE_STORED_OBJECT_ID	00.0.0.x.x.1.1	03	n [1]
SF_UNIQUE_OBJECT_ID	10.x.x.x.1.1.0	86	n [2]
SF_OPAQUE_STREAM	01.x.x.x.x.0.0	40	n [3]

1.4.2.1 Key

x = no meaning, must be zero
y = currently used in the reference implementation
n = not currently used in the reference implementation

1.4.2.2 Notes

[1] = Would allow the stored object id (stored in the CLSID field of the IStorage) to be treated as a weak reference.

[2] = Would allows unique identifiers to be stored only in the set index instead of both in the set index and a property value.
[3] = Would allow maintaining a rule that all storage elements in a file are part of an OMStorable while allowing "extra" storage elements such as the "SummaryInformation" stream.

Even though only 1 byte is needed, OMStoredForm is 2 bytes in size in order to keep each property index entry an even number of bytes in size.

Consumers must ignore index entries that they don't understand. For unknown values of _storedForm, _length is guaranteed to be valid, the bytes cannot be interpreted correctly, however they can be skipped.

1.4.3 Representations by Stored Form

Stored form name	“flat” value	“deep” value
SF_DATA	Data	None
SF_DATA_STREAM	Byte order, Stream name	IStream containing data
SF_STRONG_OBJECT_REFERENCE	Object name	IStorage containing object
SF_STRONG_OBJECT_REFERENCE_VECTOR	Vector name	IStream containing index, one IStorage per object
SF_STRONG_OBJECT_REFERENCE_SET	Set name	IStream containing index, one IStorage per object
SF_WEAK_OBJECT_REFERENCE	Tag, Key pid, Key size, Key	None
SF_WEAK_OBJECT_REFERENCE_VECTOR	Vector name	IStream containing index
SF_WEAK_OBJECT_REFERENCE_SET	Set name	IStream containing index
SF_WEAK_OBJECT_REFERENCE_STORED_OBJECT_ID	NYI	NYI
SF_UNIQUE_OBJECT_ID	NYI	NYI
SF_OPAQUE_STREAM	NYI	NYI

1.5 Capacity Limits

1.5.1 PropertyIndexHeader and PropertyIndexEntry

There is one PropertyIndexHeader per object instance. There is one PropertyIndexEntry per property instance.

PropertyIndexHeader	Field Name	Field Size	Capacity
	_byteOrder	1	'L' (little endian) or 'B' (big endian)
	_formatVersion	1	256 different revisions to the file format
	_entryCount	2	64k properties per object instance
Total		4	

PropertyIndexEntry	Field Name	Field Size	Capacity
	_pid	2	64k different property definitions per file
	_storedForm	2	64k different ways to store a property value
	_length	2	64k bytes of data per simple property
Total		6	

The capacity limits above apply only to simple property data. They do not apply to

1. streamed data such as media data (essence) and time code.
2. referenced or contained objects (singleton, vector or set)

The design allows 64k properties per object each property may be up to 64k bytes in size. That's a theoretical limit of 4096 M per object.

The design omits an _offset field from the PropertyIndexEntry and requires property values to be contiguous within the “properties” stream. This restriction could be relaxed later by assigning a _storedForm bit value to mean "unallocated and available for use".

1.5.2 Other fields

Field	Field Size	Capacity
_entryCount	4	Maximum of approximately 4 Gazillion elements in any strong/weak reference set/vector. This seems too much but it is a theoretical limit. Note that our design goal of 100,000 Mobs means that 2 bytes would be too small here. This field occurs once per collection (strong/weak reference set/vector).
_identificationSize	1	Maximum key (unique identifier) size of approximately 256 bytes. We currently have GUIDs that are 16 bytes and UMIDs that are 32 bytes. This field occurs once on each strong reference set, weak reference singleton, weak reference vector and weak reference set.
_referenceCount	4	Maximum of approximately 4 G different weak references to a given object. This field occurs on each element of a strong reference set. 0xffffffff == this object is sticky.
_referencedPropertyIndex	2	Maximum of approximately 64 k strong reference sets each containing weakly referenced objects. This field occurs once on each weak reference singleton, weak reference vector and weak reference set. It identifies the set in which the target of the weak reference(s) resides.
_localKey	4	The _localKey is the insertion key. This field is the same size as the _entryCount field.

1.6 File Signatures

[TBS]

1.7 PID Assignment

This section is informational only. It does not form part of the stored format implemented by the Object Manager. It describes the PID assignment scheme used by AAF.

PID is short for property identifier, PIDs are also known, in the AAF context, as AAF local identifiers.

1.7.1 Background

Property instances are uniquely identified by GUID. Use of a GUID allows independent, concurrent extension of the object model (users may define new optional properties) without conflict.

GUIDs are 16 bytes in size. PIDs are designed to save space. Their cost is 2 bytes per property instance instead of 16 bytes. Note that the average AAF property size is (very) approximately 10-20 bytes, making an overhead, to identify the property, of 2 bytes acceptable and an overhead of 16 bytes unacceptable.

The PIDs are unique per-file and each file has a mapping PID <-> GUID (in the dictionary). By fixing the PIDs for all predefined properties, a map is only needed for the "user defined" properties.

Note that when a user enters the definition for a new property into the dictionary they supply the GUID but not the PID. The PID is assigned automatically and transparently by the AAF implementation. The mapping PID <-> GUID for the new property is entered into the dictionary.

This means that for "user defined" properties, their PIDs are assigned and or reassigned as property instances (and, if necessary, their definitions) are moved from file to file. Thus the PIDs of "user defined" properties are not fixed and vary from file to file.

Additionally some fixed PIDs are used to "boot strap" the object model. These PIDs are used in meta-definitions.

The PID 0x0000 is reserved and is never assigned.

1.7.2 Categories of PID

Purpose	Scope	Range	Number of values	Current min - max
Never assigned	Never valid	0x0000 - 0x0000	1	
Meta-definitions	Valid and the same in all files	0x0001 - 0x00FF	255	0x0001 - 0x0020
Reserved by AAF	Valid and the same in all files	0x0100 - 0x7FFF	32512	0x0101 - 0x0315
AAF User Defined	Valid only within a particular file	0x8000 - 0xFFFF	32768	

The table shows the ranges of PIDs allocated for each purpose. Note that there may be free PIDs between the minimum and maximum assigned values.

1.7.3 Rules for PID assignment

Within a file the first PID assigned to a user defined property definition is 0xFFFF and the next is 0xFFFE and so on until the limit of 0x8000 is reached.

2. Mapping of Objects to XML

[TBS]

3. Mapping of Objects to KLV

[TBS]

4. Document History

Date	Author	Version	Change
21-Jun-01	Tim Bingham	0.1	Initial version
14-Aug-01	Tim Bingham	0.2	Add field sizes in bytes (Structured Storage)
14-Aug-01	Tim Bingham	0.3	Add section on assignment of PIDs
14-Aug-01	Tim Bingham	0.4	Add missing description of Data Streams