Advanced Authoring Format Developers' Guide

Version 1.0

Preliminary Draft

NOTICE

Product specifications are subject to change without Notice. The software described in this document is furnished under a license agreement, and may be used or copied only in accordance with the terms of the license agreement.

THE ADVANCED AUTHORING FORMAT DEVELOPERS' GUIDE IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR OR INTENDED PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE. IN NO EVENT WILL THE PROMOTERS OR ANY OF THEM BE LIABLE FOR ANY DAMAGES, INCLUDING BUT NOT LIMITED TO LOSS OF PROFITS, LOSS OF USE, INCIDENTAL, CONSEQUENTIAL, INDIRECT, OR SPECIAL DAMAGES ARISING OUT OF USE OF THIS ADVANCED AUTHORING FORMAT DEVELOPERS' GUIDE WHETHER OR NOT SUCH PARTY OR PARTIES HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.

Trademarks

Avid is a registered trademark of Avid Technology, Inc.

All other trademarks contained herein are the property of their respective owners.

Table of Contents

1. Introduction

The Advanced Authoring Format, or AAF, is an industry-driven, cross-platform, multimedia file format that will allow interchange of data between AAF-compliant applications. There are two kinds of data that can be interchanged using AAF:

- Audio, video, still image, graphics, text, animation, music, and other forms of multimedia data. In AAF these kinds of data are called ***essence*** data, because they are the essential data within a multimedia program that can be perceived directly by the audience
- Data that provides information on how to combine or modify individual sections of essence data or that provides supplementary information about essence data. In AAF these kinds of data are called ***metadata***, which is defined as data about other data. The metadata in an AAF file can provide the information needed to combine and modify the sections of essence data in the AAF file to produce a complete multimedia program.

There are two major parts to AAF:

- The *AAF Object Specification*
- The AAF Software Development Kit (SDK) Reference Implementation

The *AAF Object Specification* defines a structured container for storing essence data and metadata using an object-oriented model. The *AAF Object Specification* defines the logical contents of the objects and the rules for how the objects relate to each other. The *AAF Low-Level Container Specification* describes how each object is stored on disk. The *AAF Low-Level Container Specification* uses Structured Storage, a file storage system developed by Microsoft, to store the objects on disk.

The AAF SDK Reference Implementation is an object-oriented programming toolkit and documentation that allows client applications to access the data stored in an AAF file. The AAF SDK Reference Implementation is a platform-independent toolkit provided in source form It is also possible to create alternative implementation that accesses the data in an AAF file based on the information in the AAF Object Specification and the AAF Low-Level Container Specification.

The AAF SDK Reference Implementation provides client application a programming interface using Microsoft's Component Object Model (COM). COM provides mechanisms for components to interact independently of how the components are implemented.

The AAF SDK Reference Implementation is provided as a platform-independent source code. The AAF SDK Reference Implementation release has several reference platforms on which the SDK has been built and tested.

AAF defines a base set of built-in classes. These built-in classes can be used to interchange a broad range of data between applications, but applications may have additional forms of data that cannot be described by the basic set of built-in classes. AAF provides a mechanism to define new classes that allow applications to interchange data that cannot be described by the built-in classes.

Overview of the Files in the Source Distribution

The following are the major directories in the AAF SDK Reference Implementation source distribution:

- ref-impl — contains the source code for the reference implementation
- examples — contains C++ source examples of client applications
- tests — contains C++ source code that tests that the reference implementation is executing correctly
- doc — contains this document, the AAF Object Specification, and the AAF SDK Reference manual
- ReleaseNotes — contains build instructions, release notes, and a list of known bugs

Within the ref-impl directory, the ref-impl/include/com-api directory contains the Interface Definition Language (IDL) files for the COM interfaces available to client applications. The file AAF.idl defines the COM interfaces and methods, and the file AAFTypes.idl defines the datatypes used in the methods.

AAF Documents

The AAF documentation comprises the following:

- The *Advanced Authoring Format Object Specification* Version 1.1
- The *Advanced Authoring Format SDK Reference Implementation Methods and Interfaces* Version 1.0
- The *Advanced Authoring Format SDK Reference Implementation Developers' Guide* Version 1.0 (this document)
- The AAF Low-Level Container Specification (to be supplied)

2. Getting Started Writing AAF Client Applications

This section describes the basic concepts you need to understand to write AAF client applications. It also describes some sample interfaces and methods and includes a simple example.

Understanding Stored Objects and COM Objects

There are two kinds of objects in AAF:

0 Stored objects in AAF files

1 COM objects in memory

The client application interacts with only the COM objects in memory. The AAF SDK Reference Implementation handles the translation between the stored objects in AAF files and the COM objects in memory.

The stored objects in AAF files consist only of data; there is no code associated with a stored object in an AAF file. The COM objects in memory have both data and code. This distinction between the stored object and the COM object is required to allow alternative implementations of AAF SDK. If the stored objects in an AAF file were directly associated with code in the AAF SDK Reference Implementation, it would not be possible for an alternative implementation to use COM to access the stored object.

Not having a direct relationship between the stored object and code provides an additional benefit in that it is possible for an application to import objects defined using extensions to the built-in classes without needing to have the code that created the extensions.

Most AAF COM objects have a direct relationship to an AAF stored object, but there are some AAF COM objects that do not have a direct relationship with a single AAF stored object. These COM objects do not have a direct relationship with a stored object because they are transient objects with no stored format, such as an object created to iterate through a set of objects or because the AAF COM object model has a different level of abstraction that the AAF stored object model, for example the AAFEssenceAccess interface simplifies a set of metadata objects with a single interface.

Using the AAF COM Interfaces

Each COM object implements one or more interfaces. Each interface defines methods that allow client applications to access the data stored in the COM object and modify the object's behavior. If you are using one interface of an object and want to use another interface, you query the object to see if it supports the other interface. If the object supports the interface you requested, it returns a pointer to the interface and you can use it.

Every COM object must implement the IUnknown interface. The IUnknown interface provides the QueryInterface method, which allows you to request other interfaces from the object. In addition, IUnknown provides the AddRef and Release methods that enable reference counting on COM objects. Reference counting is a mechanism to allow separate sections of code to access an object and avoid one section of code deleting an object while another section of code still needs it. Each time a client application needs a separate access to a COM object it increments the reference count. Each time a client applications finishes use of an object, it releases it. When all sections of code that have added a reference to a COM object have released the reference, the object is deleted from memory.

In many COM object models, clients use the CoCreateInstance function provided by Microsoft's COM library, but in AAF, clients use a CreateInstance or other method provided by the AAF COM objects. The CreateInstance method creates a COM object that has a directly corresponding AAF stored object. The AAF SDK Reference Implementation uses a Dictionary to create this object.

When you create a new COM object and attach it to an AAF file, the AAF SDK Reference Implementation handles the transformations between the COM object and the stored object. Your application needs to deal only with the COM object.

When you open an existing AAF file, the AAF SDK Reference Implementation automatically creates COM objects from the stored objects.

This brief introduction to COM does not provide a sufficient understanding to create a COM client application. To learn more about COM, read one of the following books on COM:

0 *Inside COM* by Dale Rogerson, Microsoft Press, 1997

1 *Essential COM* by Don Box, Addison-Wesley, 1998

Using Global Functions to Create and Open AAF Files

To create a new AAF file, use the following global function

```
STDAPI AAFFileOpenNewModify (
             aafCharacter *  pFileName,
          aafUInt32  modeFlags,
```

```
          aafProductIdentification_t *   pIdent,
          IAAFFile ** ppFile);
```

To open an existing AAF with read-only access, use the following global function:

```
STDAPI AAFFileOpenExistingRead (
             aafCharacter *  pFileName,
             aafUInt32  modeFlags,
             IAAFFile ** ppFile);
```

To open an existing AAF file with both read and write (modify) access, use the following global function:

```
STDAPI AAFFileOpenExistingModify (
          aafCharacter *  pFileName,
          aafUInt32  modeFlags,
          aafProductIdentification_t *  pIdent,
          IAAFFile ** ppFile);
```

Creating COM Objects Using the AAF Dictionary

to be supplied

3. Exporting and Importing Essence

This chapter describes how your application can export essence to AAF files and import essence from AAF files. This chapter contains the following sections

- Importing and Exporting Basics
  - Basic algorithm for exporting essence
  - Example exporting audio data
  - Basic algorithm for importing essence
  - Example importing audio data
- Specifying the audio and video format settings
  - File Format Parameters
  - Example exporting video data
  - Example importing video data
- Accessing information about videotape, audiotape, and film
- Creating synchronized video and audio essence
- Accessing interleaved essence data

- Advanced essence import and export

Import and Export Basics

The AAF Reference Implementation provides simple interfaces to import and export audio and video data. These interfaces isolate the client application from the AAF metadata structures that describe essence data and from some of the data storage and compression details. This makes it possible for client application to process new compression formats with minimal changes. The IAAFEssenceAccess COM interface and the plugin codecs (coder/decoders) enable these simple interfaces.

The IAAFEssenceAccess interface presents a simple, uniform access mechanism for reading and writing samples of audio or video data. You access the essence data using the IAAFMasterMob interface. IAAFEssenceAccess interface hides the details of the IAAFMasterMob and the underlying IAAFSourceMob structures. For example, an IAAFMasterMob can contain an IAAFEssenceGroup structure that describes how the AAF file contains multiple versions of video data stored at different levels of compression. The IAAFEssenceAccess interface selects the version of the video data according to the criteria specified by the client. Although different formats may require different parameters to be checked or set, the methods used to read or write the data are common to all formats.

Plugin codecs read, write, compress, and decompress the video and audio data. These codecs can be dynamically loaded at runtime. This makes it possible for an application to use a hardware accelerated codec when it is available on the system. In addition, applications can import video and audio data when a codec is available even if the data is compressed using a format that was not available when the application was released.

Other interfaces allow client applications to access interleaved essence data, such as stereo audio or multi-channel MPEG or DV. Client applications can specify whether the codec should interleave/deinterleave the data or leave it to the client. In a similar manner the client application can specify whether the codec should compress/decompress the data.

The AAF Reference Implementation provides simple interfaces that make it easier to get client applications importing and exporting essence data, but it also provides interfaces that allows the client applications to access the underlying structure and data in order to perform operations not available in the simpler interfaces or to optimize access.

Basic Algorithm for Exporting Essence

The following algorithm shows the steps required to export essence data to a new AAF file.

Begin

Create a new AAF file

Get the Dictionary from the file

Create a MasterMob

Create the Essence Data specifying the codec and container

Set the properties that describe the essence data, such as sample rate

Write the samples

Set the essence to indicate you have finished writing samples

Save the AAF file

Close the AAF file

Release COM objects

End

Exporting Audio Data Example

The following example creates a new AAF file and exports audio data to it. It shows how to use the IAAFMasterMob and IAAFEssenceData COM interfaces to export audio data.

```
// First initialize Identification for Header

ProductInfo.companyName = L"AAF Development Team";
ProductInfo.productName = L"Essence Data Test";
ProductInfo.productVersion.major = 1;
ProductInfo.productVersion.minor = 0;
ProductInfo.productVersion.tertiary = 0;
ProductInfo.productVersion.patchLevel = 0;
ProductInfo.productVersion.type = kVersionUnknown;
ProductInfo.productVersionString = NULL;
ProductInfo.productID = NIL_UID;
ProductInfo.platform = NULL;

// Create a new AAF file

check(AAFFileOpenNewModify (pFileName, 0, &ProductInfo, &pFile));
check(pFile->GetHeader(&pHeader));

// Get the Dictionary from the file

check(pHeader->GetDictionary(&pDictionary));

// Create a MasterMob

// Get a Master MOB Interface
check(pDictionary->CreateInstance( &AUID_AAFMasterMob,
                  IID_IAAFMasterMob,
                  (IUnknown **)&pMasterMob));

// Get a Mob interface and set its variables.
check(pMasterMob->QueryInterface( IID_IAAFMob, (void **)&pMob));
check(pMob->GetMobID(&masterMobID));
```

```
check(pMob->SetName(L"A Master Mob"));

// Add it to the file
check(pHeader->AppendMob(pMob));

// Create the Essence Data specifying the codec, container, edit rate, and sample rate

check(pMasterMob->CreateEssence(1,           // Slot ID
                            DDEF_Sound,      // DataDefinition
                            CodecWave,       // codecID
                            editRate,        // edit rate
                            sampleRate,      // sample rate
                            kSDKCompressionDisable, // Compress disabled
                            pLocator,        // In current file
                            AAFContainer,    // In AAF Format
                            &pEssenceAccess));

// Set the properties that describe the essence data, such as audio sample size
// kAAFAudioSampleBits is stored in the audio data
   aafInt32 sampleSize = bitsPerSample;
   check(pEssenceAccess->GetEmptyFileFormat (&pFormat));
   check(pFormat->AddFormatSpecifier (kAAFAudioSampleBits,
      sizeof(sampleSize), (aafUInt8 *)&sampleSize));
   check(pEssenceAccess->PutFileFormat (pFormat));
   pFormat->Release();
   pFormat = NULL;

// Write the samples

   samplesLeft = totalNumberSamples;
   while (samplesLeft > 0) {
      check(pEssenceAccess->WriteSamples(numSamples, //Number of Samples
                                         dataPtr,    // Audio data buffer
                                         sizeof(dataBuff)));// buffer size
      samplesLeft=samplesLeft-numSamples;
   }

// Set the essence to indicate you have finished writing samples

   check(pEssenceAccess->CompleteWrite());

// Release COM interfaces

if (pEssenceAccess)
      pEssenceAccess->Release();
      pEssenceAccess = NULL;

if(pMasterMob)
      pMasterMob->Release();
   pMasterMob = NULL;
   if(pMob)
      pMob->Release();
   pMob = NULL;
```

```
    if(pDictionary)
       pDictionary->Release();
    pDictionary = NULL;
    if(pHeader)
       pHeader->Release();
    pHeader = NULL;

// Save the AAF file

    pFile->Save();

// Close the AAF file

    pFile->Close();
    pFile->Release();
    pFile = NULL;
```

The IAAFMasterMob CreateEssence method creates the IAAFEssenceAccess object and creates the following objects in the AAF file: TimelineMobSlot, File Source Mob, File Descriptor, and Essence Data.

If a Master Mob describes a single, noninterleaved stream of essence, it has one TimelineMobSlot. If a MasterMob describes video and audio data that were created from different tracks on the same source media, such as a videotape each track in the source media has a TimelineMobSlot in the MasterMob, a separate FileSourceMob and EssenceData object, and a TimelineMobSlot in the Tape SourceMob. If the essence data is interleaved, there is only a single FileSourceMob for all tracks and it has a TimelineMobSlot for each track.

If you are creating a single, noninterleaved stream of essence, you call the CreateEssence method once and there will be only one TimelineMobSlot in the MasterMob. If you are creating separate streams of essence corresponding to tracks on the source media, you call the CreateEssence method once for each track. If you are creating an interleaved stream of essence, you need to use the IAAFEssenceMultiAccess COM interface described later in this chapter.

```
HRESULT CreateEssence (
[in] aafSlotID_t  masterSlotID,
[in] aafUID_t  dataDefinition,
[in] aafUID_t  codecID,
[in] aafRational_t  editRate,
[in] aafRational_t  samplerate,
[in] aafCompressEnable_t  Enable,
[in] IAAFLocator * destination,
[in] aafUID_t  fileFormat,
[out] IAAFEssenceAccess ** access);
```

The masterSlotID parameter specifies the MobSlotID for the TimelineMobSlot in the MasterMob. The dataDefinition specifies whether the essence is sound or picture data. The codecID parameter specifies the CodecDefinition that identifies the essence format and describes the code to be used to write the essence. For example, you can specify a codec that can access WAVE audio data or one that can access AIFC audio data.

The editRate parameter specifies the units used in the TimelineMobSlot and the samplerate parameter specifies the sample rate of the essence. Often the editRate and the sampleRate are the same, but some audio TimelineMobSlots have an edit rate equal to the video sample rate. In these cases, the edit rate is not equal to the sample rate, which is the audio sample rate.

The destination and fileFormat parameters specify the name and kind of file that contains the essence data. If you are creating the essence data in the same AAF file, you can omit the destination parameter.

Basic Algorithm for Importing Essence

The following algorithm shows the steps required to import all audio essence data from an existing AAF file.

Begin

Open an AAF file

Get the Header and iterate through the MasterMobs

For each MasterMob, iterate through each MobSlot

For each MobSlot, test if it is an TimelineMobSlot describing sound data and if it is {

Open the EssenceAccess for that MobSlot

Get information about the format of the audio data

Read the samples of audio data

Release COM interfaces

}

Release COM interfaces

Close the AAF file and release its COM interface

End

Importing Audio Data Example

The following example opens an existing AAF file, finds all the audio data, and imports the audio data from it. It shows how to use the IAAFMasterMob and IAAFEssenceData COM interfaces to import audio data. Note that when you are importing audio data that was created on a system with a different byte order, the IAAFEssenceAccess interface automatically converts it to the native byte order.

```
// Open an AAF file
   check(AAFFileOpenExistingRead ( pFileName, 0, &pFile));

// Get the Header and iterate through the MasterMobs
```

```
   check(pFile->GetHeader(&pHeader));
      criteria.searchTag = kByMobKind;
      criteria.tags.mobKind = kMasterMob;
   check(pHeader->EnumAAFAllMobs(&criteria, &pMobIter));
   while(AAFRESULT_SUCCESS == pMobIter->NextOne(&pMob))
      {
      // Iterate through each MobSlot
         check(pMob->EnumAllMobSlots(&pMobSlotIter));
         while(AAFRESULT_SUCCESS == pMobIter->NextOne(&pMobSlot))
         {
         // Check to see if it is an audio timeline mob slot
HRESULT hr;
            hr=pMobSlot->QueryInterface( IID_IAAATimelineMobSlot,
               (void **) &pTimelineMobSlot);
            if (SUCCEEDED(hr) {
                  // Do not need to keep TimelineMobSlot interface
                  pTimelineMobSlot->Release();
                  pTimelineMobSlot = NULL;

                  aafUID_t SegmentDataDef;

                  check(pMobSlot->GetDataDef(&segmentDataDef))
               // Compare Segment DataDef with Sound
               if (memcmp(&segmentDataDef, &DDEF_Sound,
                        sizeof(segmentDataDef))==0)
               {
                  // Prepare to get audio data
                  // First get MobSlotID
                  check(pMobSlot->GetSlotID(&MobSlotID));
                  // Then get a Master Mob interface
                  check(pMob->QueryInterface( IID_IAAFMasterMob,
                        (void **)&pMasterMob));

                  // Open the Essence Data
                  check(pMasterMob->OpenEssence( MobSlotID,
                                    NULL,
                                    kMediaOpenReadOnly,
                                    kSDKCompressionDisable,
                                    &pEssenceAccess));

                  // Get information about the format of the audio data
                  check(pEssenceAccess->GetEmptyFileFormat (&fmtTemplate));
                  check(fmtTemplate->AddFormatSpecifier( kAAFAudioSampleBits,
                     0, NULL));
                  check(pEssenceAccess->GetFileFormat ( fmtTemplate,
                     &pFormat));
                  fmtTemplate->Release();
                  fmtTemplate = NULL;

                  check(pFormat->GetFormatSpecifier ( kAAFAudioSampleBits,
```

```
                        sizeof(sampleBits),
                        (aafDataBuffer_t)&sampleBits, &bytesRead));
                    pFormat->Release();
                    pFormat = NULL;
                    // Check we can handle this format
                    if (sampleBits==8)
                    {
                        //Read the samples of audio data
                        FinishedReading=FALSE;
                        while (!FinishedReading)
                            {
                            check(pEssenceAccess->ReadSamples(numSamples,
                                  sizeof(dataBuf),  // Maximum buffer size
                                  dataBuf,                // Buffer for the data
                                  &samplesRead,     // Actual number of samples
read
                                  &actualBytesRead));     // Actual number of
bytes read
                            if (actualBytesRead!=0)
                            {
                                  // Process audio data
                            } else FishedReading=TRUE;


      Release COM interfaces
   }
   Release COM interfaces
   Close the AAF file and release its COM interface
```

Specifying the Audio and Video Format Settings

The audio and video format parameters are stored in File Descriptors, but the IAAFEssenceAccess provides the IAAFEssenceFormat interface to get or set these parameters without having to access the File Descriptors directly.

The IAAFEssenceAccess interface defines these methods to access IAAFEssenceFormat:

- GetEmptyFileFormat—Creates an empty IAAFEssenceFormat to use in the other methods.
- PutFileFormat—Sets the File Descriptor properties to the values specified by the IAAFEssenceFormat.
- GetFileFormatParameterList—Gets all of the format parameters (without values) supported by the codec.

The IAAFEssenceFormat interface provides these methods

- AddFormatSpecifier— Adds the specified format parameter and value to an IAAFEssenceFormat
- GetFormatSpecifier— Gets the specified format parameter values. This method has an input parameter that specifies the format parameters and it creates a new IAAFEssenceFormat with the values found in the File Descriptor

- GetIndexedFormatSpecifier—Gets a format parameter by indexing through the parameters contained in the IAAFEssenceFormat

Table 3-1 lists the general, audio, and video file format parameters that can be used in IAAFEssenceFormat.

Table 3-1: Audio File Format Parameters

File Format Parameter	Type	Explanation
General Parameters		
kAAFMaxSampleBytes	aafUInt32 ???	Size of the largest sample (read only).
KAAFSampleRate	aafRational_t	Sample rate of data.
KAAFNumChannels	aafUInt32 ???	Number of channels for interleaved essence.
Audio Parameters		
kAAFAudioSampleBits	aafUInt8	Number of bits for each audio sample
Digital Image Descriptor Parameters		
kAAFFrameLayout	aafFrameLayout_t	Specifies whether all data for a complete sample is in one frame or is split into more than one field
kAAFFieldDominance		
kAAFStoredRect		
kAAFDisplayRect		
kAAFSampledRect		
kAAFPixelFormat		Specifies the format used to store the pixels, such as YUV (color difference) or RGBA.
kAAFAspectRatio		
kAAFAlphaTransparency		
kAAFGamma		
kAAFImageAlignmentFactor		
kAAFVideoLineMap		
kAAFLineLength		
kAAFWillTransferLines		
CDCI Parameters		
kAAFCDCICompWidth		
kAAFCDCIHorizSubsampling		
kAAFCDCIColorSiting		
kAAFCDCIBlackLevel		
kAAFCDCIWhiteLevel		

kAAFCDCIColorRange
kAAFCDCIPadBits
kAAFPixelSize
RGBA Parameters
kAAFRGBCompLayout
kAAFRGBCompSizes
kAAFRGBPalette
kAAFRGBPaletteLayout
kAAFRGBPaletteSizes

Exporting Video Data Example

The following example creates a new AAF file and exports uncompressed RGBA video data to it. It shows how to use the IAAFMasterMob and IAAFEssenceData COM interfaces to export video data.

```
// First initialize Identification for Header

ProductInfo.companyName = L"AAF Developers Desk";
ProductInfo.productName = L"Essence Data Test";
ProductInfo.productVersion.major = 1;
ProductInfo.productVersion.minor = 0;
ProductInfo.productVersion.tertiary = 0;
ProductInfo.productVersion.patchLevel = 0;
ProductInfo.productVersion.type = kVersionUnknown;
ProductInfo.productVersionString = NULL;
ProductInfo.productID = NIL_UID;
ProductInfo.platform = NULL;

// Create a new AAF file

check(AAFFileOpenNewModify (pFileName, 0, &ProductInfo, &pFile));
check(pFile->GetHeader(&pHeader));

// Get the Dictionary from the file

check(pHeader->GetDictionary(&pDictionary));

// Create a MasterMob

// Get a Master MOB Interface
check(pDictionary->CreateInstance( &AUID_AAFMasterMob,
                  IID_IAAFMasterMob,
                  (IUnknown **)&pMasterMob));

// Get a Mob interface and set its variables.
```

```
check(pMasterMob->QueryInterface(IID_IAAFMob, (void **)&pMob));
check(pMob->GetMobID(&masterMobID));
check(pMob->SetName(L"A Master Mob"));

// Add it to the file
check(pHeader->AppendMob(pMob));
```

// **Create the Essence Data specifying the codec, container, edit rate, and sample rate**

```
// set edit and sample rate to 30000/1001
check(pMasterMob->CreateEssence(1,           // Slot ID
                          DDEF_Picture,           // DataDefinition
                          CodecRBBA,        // codecID
                          editRate,         // edit rate
                          sampleRate,       // sample rate
                          kSDKCompressionDisable, // Compress disabled
                          pLocator,         // In current file
                          AAFContainer,     // In AAF Format
                          &pEssenceAccess));
```

// **Set the properties that are common to all video formats**

```
   check(pEssenceAccess->GetEmptyFileFormat (&pFormat));
   check(pFormat->AddFormatSpecifier (kAAFFrameLayout,
      sizeof(kOneField), kOneField));
   check(pFormat->AddFormatSpecifier (kAAFStoredRectangle,
      TBS));
```

// **Set the properties that are specific to the RGBA format**

```
      check(pFormat->AddFormatSpecifier (kAAFRGBAxxxx,
      TBS));
```

// **Set the EssenceAccess to the specified file formats**

```
check(pEssenceAccess->PutFileFormat (pFormat));
   pFormat->Release();
   pFormat = NULL;
```

// **Write the samples**

```
   samplesLeft = totalNumberSamples;
   while (samplesLeft > 0) {
      check(pEssenceAccess->WriteSamples(numSamples, //Number of Samples
                                         dataPtr,    // data buffer
                                         sizeof(dataBuff)));// buffer size
      samplesLeft=samplesLeft-numSamples;
   }
```

// **Set the essence to indicate you have finished writing samples**

```
   check(pEssenceAccess->CompleteWrite());
```

// **Release COM interfaces**

```
   if(pMasterMob)
      pMasterMob->Release();
   pMasterMob = NULL;
   if(pMob)
      pMob->Release();
   pMob = NULL;

   if(pDictionary)
      pDictionary->Release();
   pDictionary = NULL;
   if(pHeader)
      pHeader->Release();
   pHeader = NULL;

// Save the AAF file

   pFile->Save();

// Close the AAF file

   pFile->Close();
   pFile->Release();
```

To modify this program to export JPEG video data, the following are the only modifications required:

```
// Create the Essence Data specifying the codec, container, edit rate, and sample rate

   // set edit and sample rate to 30000/1001
   check(pMasterMob->CreateEssence(1,           // Slot ID
                              DDEF_Picture,           // DataDefinition
                              CodecJPEG,        // codecID
                              editRate,         // edit rate
                              sampleRate,       // sample rate
                              kSDKCompressionEnable,  // Compress disabled
                              pLocator,         // In current file
                              AAFContainer,     // In AAF Format
                              &pEssenceAccess));

// Set the properties that are specific to the RGBA format
      check(pFormat->AddFormatSpecifier (kAAFCDCIxxxx,
      TBS));
```

Importing Video Data

to be supplied

Accessing information about Videotape, Audiotape, and Film

to be supplied

Creating Synchronized Video and Audio

The Master Mob synchronizes audio and video data. You add the video and audio data as separate TimelineMobSlots in the MasterMob. For example, if the video and audio data was digitized from a videotape source with one video track and two audio tracks, the Master Mob should contain one video TimelineMobSlot and two audio TimelineMobSlots.

It is important to set the PhysicalTrackNumber especially when you are exporting stereo audio data. The left channel should have a PhysicalTrackNumber of 1 and the right channel should have a PhysicalTrackNumber of 2.

If you are exporting synchronized audio and video data to an AAF file, the reference implementation does not ensure that the essence data is contiguous. For example, if you alternate writing video and audio data, the essence data may be broken into chunks. This may reduce efficiency when reading the data. There are three ways to increase the probability of writing contiguous data:

- Write each complete essence data sequentially, completing the writing of one essence data before you start the next one. This method is impractical if you are aquiring different tracks of essence at the sample rate.
- Write each stream to a separate AAF file or to separate nonAAF data files.
- Preallocate the storage for each separate essence data.

Accessing Interleaved Essence Data

to be supplied

Advanced Features for Essence Export and Import

Exporting and Importing Essence Data Stored in Other Containers

The client application needs only minor changes to import or export Essence Data that is described by an AAF file but is stored in another container. In order to export essence data in another container, such as an ordinary file, you must specify the container format and the locator in the IAAFMasterMob->CreateEssence method.

If you are importing essence data stored in another container format and the IAAFMasterMob->OpenEssence method can find the container using the locator, the client application does not need to do anything else. If the OpenEssence method cannot find the container using the locator, you can try to find the container by using the xxxxx.

Reading and Writing Unprocessed Essence Data

to be supplied

Accessing the File Descriptor Directly

To access properties in the File Descriptor that are not accessible through the IAAFEssenceFormat interface, you get the File Descriptor and set or get its properties directly. Typically, you only need to do this if your application uses extensions in the File Descriptor or if you are using a File Descriptor that is not supported by the AAF Reference Implementation.

You can get the File Descriptor by using the IAAFSearchSource interface to find the File Source Mob. Then you use the IAAFSourceMob->GetFileDescriptor method to get the IAAFFileDescriptor interface. For example:

```
   check(pMasterMob->OpenEssence(MobSlotID,
         NULL,
         kMediaOpenReadOnly,
         kSDKCompressionDisable,
         &pEssenceAccess));
   check(pMob->FindSlotBySlotID (MobSlotID,
      &pMobSlot);
check (pMobSlot->GetOrigin(&origin);
check (pMasterMob->SearchSource(MobSlotID, origin,
      kFileMob,
      NULL, // specify the same essence criteria as Open Essence
      &sourceInfo);
check(sourceInfo->GetMob(&pMob);
check(pMob->QueryInterface(IID_IAAFSourceMob, (void **)&pSourceMob));
check(pSourceMob->GetEssenceDescriptor(&pEssenceDescriptor));
```

4. Exporting Compositions

Exporting a Simple Composition

This example opens an existing AAF file for modification and creates a new Composition Mob in it. The Composition Mob contains a Sequence that has a Source Clip for each with one audio Mob Slot that contains a Sequence containing Source Clips. This example assumes that the Master Mobs in the existing file contains audio TimelineMobSlots with an 48MHz edit rate.

```
aafRational_t editRate;

editRate.numerator=48000;
editRate.denominator=1;

// Open an existing AAF file for modification

check(AAFFileOpenExistingModify (pFileName, 0, &ProductInfo, &pFile));
check(pFile->GetHeader(&pHeader));

// Get the Dictionary from the file

check(pHeader->GetDictionary(&pDictionary));

// Create a Composition Mob
check(pDictionary->CreateInstance( &AUID_AAFCompositionMob,
                  IID_IAAFMob,
```

```
                  (IUnknown **)&pCompMob));
// Add it to the file
check(pHeader->AppendMob(pCompMob));


// Create a TimelineMobSlot with an audio sequence
// First create a Sequence and initialize it
check(pDictionary->CreateInstance( &AUID_AAFSequence,
         IID_IAAFSequence,
         (IUnknown **)&pAudioSequence));
check(pAudioSequence->Initialize(DDEF_Sound));
// Then get the Segment interface and use it to append a new
// Timeline Mob Slot
check(pAudioSequence->QueryInterface (IID_IAAFSegment, (void **)&seg));
check(pCompMob->AppendNewTimelineSlot (editRate, seg, 1, slotName,
               &newSlot));

// Now we need to iterate through all Master Mobs in the file
criteria.searchTag = kByMobKind;
criteria.tags.mobKind = kMasterMob;
check(pHeader->EnumAAFAllMobs(&criteria, &pMobIter));

while((AAFRESULT_SUCCESS == pMobIter->NextOne(&pMob)))
{
   // Find the first MobSlot with a Sound data definition
   lookingForAudio=true;
   check(pMob->EnumAAFAllMobSlots(&pMobSlotIter));

   // Iterate through each MobSlot

   while((AAFRESULT_SUCCESS == pMobSlotIter->NextOne(&pMobSlot)) &&
            lookingForAudio);
   {

      // Check to see if it is an audio timeline mob slot
         HRESULT hr;
         AafUID_t segmentDataDef;
         IAAFTimelineMobSlot *pTimelineMobSlot = NULL;
         // First check that the MobSlot is a TimelineMobSlot
         hr=pMobSlot->QueryInterface(IID_IAAATimelineMobSlot,
            (void **) &pTimelineMobSlot);
         if (SUCCEEDED(hr) {
            // it is a TimelineMobSlot
            // Now check if it has a sound segment
            check(pMobSlot->GetDataDef(&segmentDataDef))
            // Compare Segment DataDef with Sound
            if (memcmp(&segmentDataDef, &DDEF_Sound,
                  sizeof(segmentDataDef))==0
            {
               // It has sound, now get info for new Source Clip
               aafSourceRef_t sourceRef;
               check(pMob->GetMobID(sourceRef.sourceID));
               check(pMobSlot->GetSlotID(sourceRef.sourceSlotID));
```

```
                check(pTimelineMobSlot->GetOrigin(sourceRef.startTime));
                check(pMobSlot->GetSegment(&pSegment);
                check(pSegment->QueryInterface(IID_IAAFComponent,
                   (void **)&pComponent));
                check(pComponent->GetLength(&duration));

                // Create new Source Clip
                check(pDictionary->CreateInstance( &AUID_AAFCSourceClip,
                   IID_IAAFSourceClip,
                   (IUnknown **)&pSourceClip));
                // Initialize Source Clip
                check(pSourceClip->Initialize( (aafUID_t*) &DDEF_Sound,
                                   duration, sourceRef));
                check(pSourceClip->QueryInterface (IID_IAAFComponent,
                                   (void **) &pComponent));
                check(pAudioSequence->AppendComponent(pComponent));
                // Stop looking for audio in this Master  Mob
                lookingForAudio = false;


            }

        }

                // finish up

    }
```

Exporting a Composition with Transitions and Operation Groups

```
// Create a new AAF file

   check(AAFFileOpenNewModify (pFileName, 0, &ProductInfo, &pFile));
   check(pFile->GetHeader(&pHeader));

   // Get the Dictionary from the file

   check(pHeader->GetDictionary(&pDictionary));
   // Create a Composition Mob
   check(pDictionary->CreateInstance( &AUID_AAFCompositionMob,
                    IID_IAAFMob,
                    (IUnknown **)&pCompMob));

   // Create a TimelineMobSlot with an audio sequence
   check(pDictionary->CreateInstance( &AUID_AAFSequence,
            IID_IAAFSequence,
            (IUnknown **)&pAudioSequence));
   check(pAudioSequence->QueryInterface (IID_IAAFSegment, (void **)&seg));
```

```
check(pAudioSequence->QueryInterface( IID_IAAFComponent,
                    (void **)&aComponent));

check(aComponent->SetDataDef(&audioDef));
check(pCompMob->AppendNewTimelineSlot ( editRate, seg, 1, slotName,
               &AudioSlot));
// Create a TimelineMobSlot with an video sequence
check(pDictionary->CreateInstance( &AUID_AAFSequence,
         IID_IAAFSequence,
         (IUnknown **)&pVideoSequence));
check(pVideoSequence->QueryInterface ( IID_IAAFSegment, (void **)&seg));

check(pVideoSequence->QueryInterface( IID_IAAFComponent,
                    (void **)&aComponent));

check(aComponent->SetDataDef(&videoDef));
check(pCompMob->AppendNewTimelineSlot ( editRate, seg, 2, slotName,
               &videoSlot));

// Create new Source Clips and MonoAudioMixdown and append to audio TimelineMobSlots
check(pDictionary->CreateInstance( &AUID_AAFCSourceClip,
               IID_IAAFSourceClip,
               (IUnknown **)&pSourceClip));
// Initialize Source Clip
check(pSourceClip->Initialize( SOUND_DataDef,
                    sourceRefA1, duration1);
check(pSourceClip->QueryInterface ( IID_IAAFComponent,
                                 (void **) &pComponent));
check(pAudioSequence->AppendComponent( pComponent));

// Create two Source Clips for Mixdown

check(pDictionary->CreateInstance( &AUID_AAFCSourceClip,
               IID_IAAFSourceClip,
               (IUnknown **)&pSourceClipM1));
check(pSourceClip->Initialize( SOUND_DataDef,
                    sourceRefM1, durationM);
check(pSourceClip->QueryInterface ( IID_IAAFSegment,
                                 (void **) &pSegmentM1));
check(pDictionary->CreateInstance( &AUID_AAFCSourceClip,
               IID_IAAFSourceClip,
               (IUnknown **)&pSourceClipM2));
check(pSourceClip->Initialize( SOUND_DataDef,
                    sourceRefM1, durationM);
check(pSourceClip->QueryInterface ( IID_IAAFSegment,
                                 (void **) &pSegmentM2));

// Get or create the OperationDefinition

check(getAAFOperationDefinition( AAF_AUD_Mixdown, &pOpDef));

check(pDictionary->CreateInstance( &AUID_AAFCOperationGroup,
```

```
                IID_IAAFOperationGroup,
                (IUnknown **)&pOperationGroup));
    // Initialize Operation Group
    check(pOperationGroup->Initialize( Sound_DataDef,
                        durationM, pOpDef);
    // Append the two audio Source Clips to the  mixdown
    check(pOperationGroup->AppendSegment(pSegmentM1));
    check(pOperationGroup->AppendSegment(pSegmentM2));
    // no parameters needed for mixdown
    // append mixdown to Sequence
    check(pOperationGroup->QueryInterface ( IID_IAAFComponent,
                                    (void **) &pComponent));
    check(pAudioSequence->AppendComponent( pComponent));

    // Append Source Clips and dissolve Transition to video  TimelineMobSlot

    // First a Source Clip before the Transition

    check(pDictionary->CreateInstance( &AUID_AAFCSourceClip,
                IID_IAAFSourceClip,
                (IUnknown **)&pSourceClip));
    // Initialize Source Clip
    check(pSourceClip->Initialize( Picture_DataDef,
                        sourceRefV1, durationV1);
    check(pSourceClip->QueryInterface ( IID_IAAFComponent,
                                    (void **) &pComponent));
    check(pVideoSequence->AppendComponent( pComponent));

    // Before we create the Transition, we need the  OperationGroup
    // Get or create the OperationDefinition

    check(getAAFOperationDefinition( AAF_Video_Dissolve, &pOpDef));

    check(pDictionary->CreateInstance( &AUID_AAFCOperationGroup,
                IID_IAAFOperationGroup,
                (IUnknown **)&pOperationGroup));
    // Initialize Operation Group
    check(pOperationGroup->Initialize( Picture_DataDef,
                        durationT, pOpDef);
    // Since its used in a Transition,  OperationDefinition has not Segments
    cutPoint=durationT/2;  // Set the cutPoint to midway through Transition

    // Then create a Transition and append it to the Sequence

    check(pDictionary->CreateInstance( &AUID_AAFCTransition,
                IID_IAAFTransition,
                (IUnknown **)&pTransition));
    // Initialize Transition
    check(pTransition->Initialize( Picture_DataDef,
                        durationT,cutPoint, pOperationGroup));
    //Append Transition to video Sequence
```

```
   check(pTransition->QueryInterface ( IID_IAAFComponent,
                                       (void **) &pComponent));
   check(pVideoSequence->AppendComponent( pComponent));

   // Then append a Source Clip after the Tran sition

   check(pDictionary->CreateInstance( &AUID_AAFCSourceClip,
                  IID_IAAFSourceClip,
                  (IUnknown **)&pSourceClip));
   // Initialize Source Clip
   check(pSourceClip->Initialize(  Picture_DataDef,
                         sourceRefV2, durationV2);
   check(pSourceClip->QueryInterface ( IID_IAAFComponent,
                                       (void **) &pComponent));
   check(pVideoSequence->AppendComponent( pComponent));

// ********


   check(pVideoSequence->AppendComponent( pComponent));
               rc = pDictionary->CreateInstance(&AUID_AAFOperationGroup,
IID_IAAFOperationGroup, ( IUnknown **) &pEffect);
                  rc =
GetAAFOperationDefinition("omfi::effectSimpleMonoAudioDissolve", "Simple Mono
Audio Dissolve", "Combines two mono audio streams",
                                       -1, AAFFalse, 2, DDEF_Sound, &pEffectDef);
                  rc = GetParameterDefinition((aafUID_t
*)&kAAFParameterDefLevel, NULL,
                                                          L"Level",
                                                          L"Level, equal to mix
ratio of B/A. Range is 0 to 1. The formula  P = ( Level*B)+((1-Level)*A)",
                                                          L" ",
                                                          &pParameterDef);
                  pEffectDef->AddParameterDefs( pParameterDef);
                  pEffect->Initialize(&datadef, (aafLength_t)OMFLength,
pEffectDef);
                  pEffect->SetBypassOverride(-1);
                  pEffect->QueryInterface( IID_IAAFComponent, (void
**)ppComponent);
                  pEffect->Release();
                  pEffectDef->Release();
                  pParameterDef->Release();
```

Here is the example code for getAAFOperationDefinition, which checks if the operation definition exists and appends the definition to the dictionary, if needed.

```
HRESULT getAAFOperationDefinition(
            aafUID_t effectDefAUID,
            IAAFOperationDef ** ppOpDef);
{
   // Look in the dictionary to find if the effect Definition exists
   // if it exists use it.
   rc = pDictionary->LookupOperationDefinition(& effectDefAUID, ppOpDef);
```

```
   if (FAILED(rc))
   {
      pDictionary->CreateInstance(&AUID_AAFOperationDef, IID_IAAFOperationDef,
         (IUnknown **) ppOpDef);
      (*ppEffectDef)->QueryInterface( IID_IAAFDefObject, (void **)&pDefObject);
      pDefObject->Init(&effectDefAUID, pwName, pwDesc);
      pDefObject->Release();
      pDefObject = NULL;
      if (!memcmp(effectDefAUID, kAAFEffectVideoDissolve,
         sizeof[kAAFEffectVideoDissolve]) {
         // Define video dissolve effect

      } elseif (!memcmp(effectDefAUID, kAAFEffectVideoDissolve,
         sizeof[effectDefAUID]) {
         // define xxx effect
      } else
      {
         // unknown effect
      }
      pDictionary->RegisterOperationDefinition(*ppEffectDef);
      (*ppEffectDef)->SetIsTimeWarp((aafBool)isTimeWarp);
      (*ppEffectDef)->SetCategory(pwName);
      (*ppEffectDef)->SetBypass((aafUInt32 )bypassOverride);

      (*ppEffectDef)->SetNumberInputs(numberInputs);
      (*ppEffectDef)->SetDataDefinitionID(&defDataDef);

      rc = AAFRESULT_SUCCESS;
   }
}

HRESULT getAAFParameterDefinition(
            aafUID_t parameterDefAUID,
            IAAFParameterDef ** ppParDef);
{
   // Look in the dictionary to find if the parameter Definition exists
   // if it exists use it.
   rc = pDictionary->LookupParameterDefinition(&parameterDefAUID, ppParDef);
   if (FAILED(rc))
   {
      if (!memcmp(effectDefAUID, kAAFEffectVideoDissolve,
         sizeof[kAAFEffectVideoDissolve]) {
         // Define video dissolve effect

      } elseif (!memcmp(effectDefAUID, kAAFEffectVideoDissolve,
         sizeof[effectDefAUID]) {
         // define xxx effect
      } else
      {
         // unknown effect
      }
```

```
   if (newParameterDefined) {
   pDictionary->CreateInstance(&AUID_AAFParameterDef, IID_IAAFParameterDef,
      (IUnknown **) ppParDef);
      (*ppParDef)->QueryInterface( IID_IAAFDefObject, (void **)&pDefObject);
      pDefObject->Init(&parameterDefAUID, pwName, pwDesc);
      pDefObject->Release();
      pDefObject = NULL;
   pDictionary->RegisterParameterDefinition(*ppParDef);
      (*ppEffectDef)->SetDataDefinitionID(&defDataDef);
      rc = AAFRESULT_SUCCESS;
   }
   }
}
```

5. Extending AAF

Overview of Extending AAF

The Advanced Authoring Format is designed to allow extensions. AAF files can include extensions that define new effects, new kinds of metadata, and new kinds of essence data.

You may want to extend AAF in order to:

- Add optional properties to built-in classes to interchange some supplementary information
- Define new classes to interchange new kinds of metadata
- Define new kinds of effects
- Define new kinds of audio or video compression
- Define new kinds of essence
- Describe plug-in code that can process essence or effects or provide behavior for new classes
- Define new container mechanisms to store essence data

As the technologies of authoring applications advance, people can use the applications to do new things and will want to interchange this new kind of information between applications. Typically, these new features are added by one or a few applications, and gradually, as the technology matures, the features become common to many applications. Consequently, these features are first defined as private

extensions and may later progress to be included in the AAF Object Specification and directly supported by the AAF Reference Implementation SDK. Features that are included in the AAF Object Specification and the AAF Reference Implementation are described in this chapter as *built-in* features in contrast to the extended features that you can add to AAF.

> *Note* Examples in this chapter are shown in either pseudo-code or as C code using COM interfaces.

Adding Definitions to the Dictionary

To extend AAF, you first add definitions for the new item to the dictionary in an AAF file, then you can use the new definition to create the items with the extended data. You can create definitions using the CreateInstance method in the same manner that you use it to create AAF objects in Mobs. Alternatively, you can create the definitions once in a definitions-only AAF file and then clone the definitions in this file to new files that you create.

All extensions that are used in an AAF file are defined in the dictionary. The information in the dictionary makes it easier for an application to preserve extended information that has been created by other applications.

Defining New Effects

To define a new effect, you add an EffectDefinition to the dictionary. If the effect uses new kinds of parameters, you also need to add ParameterDefinitions to the dictionary. You can optionally describe plug-in code that makes it easier to process the effect or that can be used to perform the effect. In addition, you can describe interpolation plugins that calculate parameters for intermediate samples based on the values at key samples.

Defining New Classes and Properties

To define a new class, you add a ClassDefinition to the dictionary, then you can use the ClassDefinition to create objects with the new class. You can add optional properties to an existing class or define properties for a new class by adding PropertyDefinitions to the dictionary. If you define a new class, it must have one of the AAF built-in classes as a superclass.

Defining New Datatypes

When adding optional properties to built-in classes or defining properties in new classes, you can use the built-in data types or you can define new data types. New data types are defined using data type building blocks that allow you to describe the structure of the new data types. For example, if you want to define a property that specifies a color by giving 4 integer values for red, green, blue, and alpha, you can define it as a fixed array with four integer elements.

Defining New Essence

You can extend AAF so that it can describe and encapsulate new kinds of essence. The new essence can be a new compression for video or audio or it can represent an entirely new form of data, such as motion sensing data that can be used as a basis of animating 3D objects.

Describing Plug-in Code

The AAF Reference Implementation is intended for data interchange between applications, which may be running on different platforms and which may have different mechanisms for displaying essence and for processing effects. Consequently, the most important goal of the AAF extension mechanism is to enable applications to interchange a broad variety of data. In order to interchange data, it may be necessary to identify code that is needed to the process the data. AAF provides a mechanism to describe plug-in code that can be used to process essence data or metadata.

Plugins are described by adding PluginDescriptions to the dictionary. A DefinitionObject, such as an EffectDefinition, ClassDefinition, or CodecDefinition, identifies the plug-in code that can be used with it. Since there are several different APIs that plug-in code can support, AAF provides a mechanism for the client application to query the plug-in code. PluginDescriptions have a set of locators that identify addresses where the plug-in code may be found. These addresses may be local to the user's computer or local area network or may be a remote location specified by an internet URL.

In most cases, it is the application's responsibility to define the kind of plug-in interface and how the plugin is dynamically linked and executed. In these cases the application uses the AAF reference implementation SDK to discover the identity of the plug-ins that are available to process data and then the application invokes the plug-in.

Application can interchange data described by new classes either with plug-in code that accesses the new classes or without plug-in code. Applications can use the property direct interfaces to access the new classes. As a convenience the developers of the new class can provide plug-in code that makes it easier to access the new classes.

Recommended Practices for Extending AAF

Since the key goal of AAF is to allow users to interchange information between applications, it is important that extensions be made in ways that meet the following goals, listed in order starting with the highest priority goals.

- Ensure that the user does not get wrong or misleading data
- Define extensions in a way that does not interfere with the interchange of the metadata described by the built-in features of AAF
- Share extended data between applications, when appropriate
- Preserve extended data, whenever possible

Although it is important to share and preserve extended data, it is more important to avoid giving the user wrong or misleading data. Although it would be ideal to be able to interchange all extended data, it is better to inform the user that some data cannot be interchanged than to misinterpret data without notifying the user. If the user is warned that some data is being lost, he or she can take corrective action. But if the interchange seems to work but has produced errors, the user may continue with the production process and not discover the error until later when it may be more costly or impossible to correct.

To meet these goals, you should follow these practices when extending AAF:

- Add optional properties to a built-in class or define subclasses of concrete built-in classes only if other applications can ignore the extensions without producing erroneous data
- If you are defining supplementary information, when possible you should define it as optional properties on built-in classes rather than defining new subclasses
- If you are defining new compression formats or mechanisms for multiplexing different kinds of essence, provide plug-in code that enable applications to access the data
- If your application is not coded to understand an extension, preserve the extended data whenever feasible

Adding DefinitionObjects to the Dictionary

Each DefinitionObject has the following properties:

Identification: AUID
Name: String
Description: String

The Identification property provides a unique integer that identifies the definition. The Name property provides a name for display purposes. To ensure that property names do not conflict, the Name should consist of a qualified string, where the fields are separated by : (colon characters). The Description provides a brief explanation of the purpose of the object.

In addition, a DefinitionObject can have an array of weak references to PluginDescriptors, which identify code that can be used to process items that reference the definition.

Objects can use DefinitionObjects by having a weak reference to them. For example, a Group object in a Mob has a weak reference to an Effect Definition. A DefinitionObject in the dictionary can have a weak reference to another DefinitionObject. For example, an EffectDefinition object a set of weak references to the ParameterDefinitions that define the parameters that can be used for that effect.

Defining New Effects

The EffectsDefinition class defines new effects. Effect definitions include the following:

- AUID that identifies the effect
- Effect name and description for display purposes
- Plugin descriptors
- Number of essence input segments, specifies -1 for effects that can have any number of input essence segments
- Set of weak references to ParameterDefinitions that specify parameters that may be used with the effect. Each ParameterDefinition specifies
    - ◊ AUID identifying parameter

◊ Weak reference to a DatatypeDefinition, which specifies the effective type of the Parameter

When appropriate new Effect Definitions should use existing parameters and data kinds. If an Effect Definition specifies a previously defined parameter, it must specify the same data kind.

Adding Optional Properties to Built-In Classes

To add optional properties to built-in classes, do the following:

```
Use Dictionary->LookupClass to get the ClassDefinition for the built-in class
Use Dictionary->CreateInstance to create new PropertyDefinition objects
For each new property
(
    Either
        (
            (
                Use Dictionary->CreateInstance to create the new TypeDefinition
                Use TypeDefinition->Initialize to set the values of the TypeDefinition
            )
    or
        (
            Use Dictionary->LookupType to get an existing TypeDefinition
        )
    )
    Use PropertyDefinition->Initialize to set the Identification, Name, Description, TypeDefinition,
and
        optional
    Use ClassDefinition->AddProperty to add the property to the ClassDefinition
    Use AAFObject->SetPropertyValue to set the optional property
    Use AAFObject->GetPropertyValue to get the optional property

)
```

Defining New Classes

You can define new classes with or without plugin code. If you define a new class without plugin code, both the application that defines the new class and applications that use it use the property direct interfaces to access the properties. If you define a new class and provide plugin code, then the application that defines the new class can use the property access interfaces of the plugin code to access the properties. An application using a new class that has a plugin available can either use the plugin to access the properties or can use the property direct interfaces to access them.

In order to understand how plugins work with new classes, it is necessary to understand how the reference implementation creates run-time objects using the stored objects in an AAF file. The reference implementation has an object manager that creates an implementation object from an object stored in an AAF file. Client applications directly access only the implementation object. The reference implementation automatically creates the implementation object from the stored object and automatically

creates or updates the stored object based on the implementation object. The client application does not directly access the stored object. The client application is responsible for calling the AAFFile->Save method before calling the AAFFile->Close method to ensure that the values in the stored object are set to those specified by the corresponding implementation objects.

Figure A illustrates how the object manager in the reference implementation translates between the implementation object that the client accesses and the stored object in the AAF file. The reference implementation creates the implementation object, when the client application makes the following call to CreateInstance

```
Dictionary->CreateInstance (&AUID_AAFSourceClip,
                            IID_IAAFSourceClip,
                            (IUnknown **)&Sclp))
```

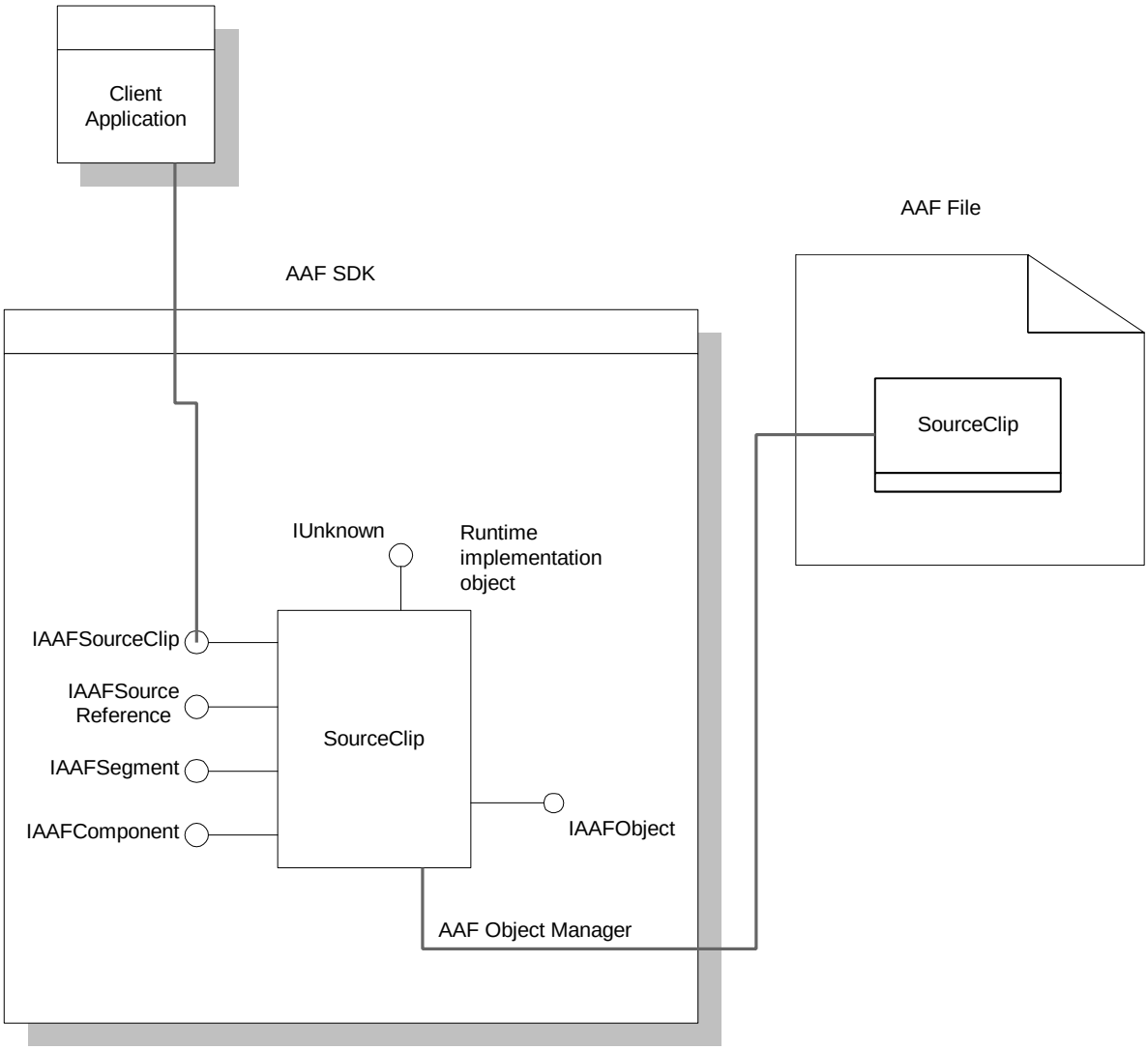

Figure A: Stored Objects in File and Run-time Implementation Objects

The reference implementation creates the AAFSourceClip implementation object when either the Dictionary->CreateInstance method is called with AUID of the stored SourceClip or when the client application follows a strong reference property to a SourceClip object.

When the client application calls the AAFFile->Save method, the reference implementation writes the property values of the runtime implementation object to the stored object in the AAF file.

For built-in classes, there is a one-to-one correspondence between the stored object class and the implementation object class.

But for extended classes, the reference implementation creates a different kind of implementation object depending on whether there is plugin code available when it creates the implementation object. If there is plugin code available for the new class, the reference implementation creates an implementation object using the plugin code. If there is no plugin code available, the reference implementation creates an implementation object using the built-in object that corresponds to the most direct superclass that is a built-in class. Figure B illustrates the relationship between stored object and implementation object when there is no plugin code available and Figure C illustrates this relationship when there is plugin code available.

When there is no plugin code, an application makes the following call to CreateInstance and the reference implementation creates an implementation object for the built-in SourceClip object to correspond to a stored object with a new class that is a subclass of SourceClip (as illustrated in Figure B).

```
Dictionary->CreateInstance (&AUID_ExtendedSourceClip,
        IID_IAAFSourceClip,
        (IUnknown **)&Sclp))
```

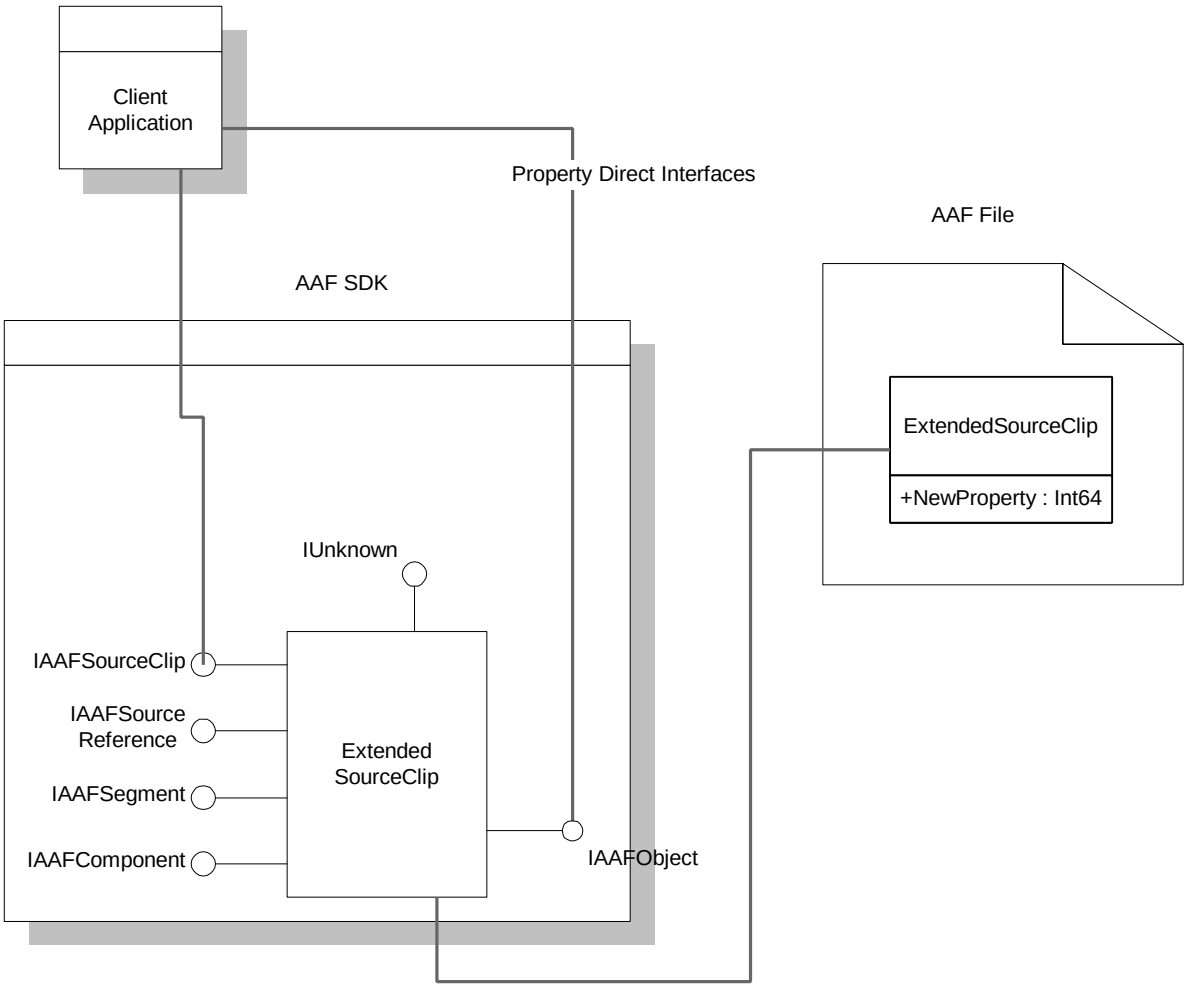

Figure B: Stored Objects and Implementation Object for New Classes without Plugin

When there is plugin code, an application makes the following call to CreateInstance and the reference implementation creates an implementation object specified by the plugin to correspond to a stored object with the new class, as illustrated in Figure C.

```
Dictionary->CreateInstance (&AUID_ExtendedSourceClip,
                IID_IExtendedSourceClip,
                (IUnknown **)&Sclp))
```

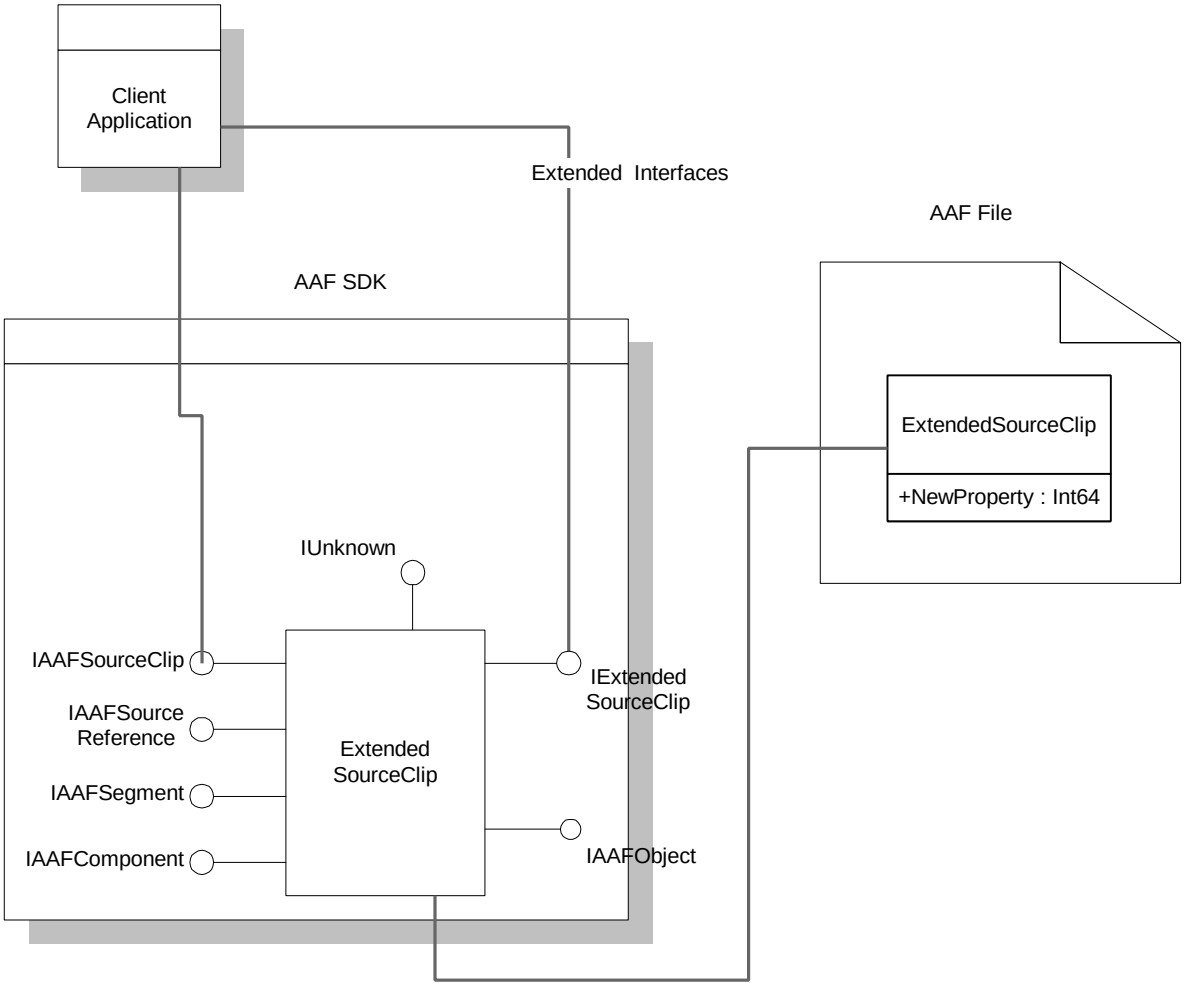

Figure C: Stored Objects and Implementation Object for New Classes with Plugin

Defining New Datatypes

You define a new datatype by using the built-in datatype building blocks. All of the built-in datatypes are defined by using the same building blocks. These building blocks provide a mechanism to describe datatypes so that the AAF Reference Implementation and applications can perform some basic operations on properties with extended datatypes. These datatype definitions make it possible for applications to preserve extensions even if they do not recognize the datatype. The datatypes building blocks comprise a small set of basic datatypes and a small set of datatype tools that allow you to combine other datatypes to define more complex types.

The basic datatypes are:

- Integers
- Strong and weak object references
- Streams, which are used to store essence data and other very large chunks of data

The datatype tools that combine other data types are:

- Fixed or varying length arrays (ordered)
- Unordered set (typically used with object references)
- Records consisting of named subfields, each with a specified datatype
- Enumerated types, which specify allowed sets of values
- Renamed types, which share the same structure as the underlying types but have separate identification to provide meaning or enable type checking

Integers and object references are the basic types that define the basic leaves of more complex types. Although some data, such as text strings, may not have a meaning specified by integers, it can usually be processed as a structure built from integer components. For example, a wide-character string can be defined as the following:

```
Define Wide-character-string as
    String of 16-bit integers
```

The built-in Rational type can be defined as:

```
Define Rational as
    Record of
    (
        Numerator: 32-bit integer,
        Denominator: 32-bit integer
    )
```

The built-in FrameLayout type can be defined as:

```
Define FrameLayout as
    Enumerate 16-bit integer with values:
    (
        0:  “FULL_FRAME”,
        1:  “SEPARATE_FIELDS”,
        2:  “SINGLE_FIELD”,
        3:  “MIXED_FIELDS”
    )
```

Creating Plugins

You can define plug-in code that perform tasks such as:

- Provide access helper interfaces for new class definitions
- Provide code to compress or convert essence data
- Provide code to retrieve and store essence data that in container formats such as asset management systems
- Provide code to access effects in Transitions and Group objects or to perform the effect using the input segments
- Provide code to interpolate parameter values when performing an effect

Defining New Kinds of Compression or Essence

The scope of the task of defining new essence types varies greatly depending on how different the new essence type is from the existing ones. Defining a new essence type can consist of any of the following

- Defining a new compression method for an existing data kind, such as video
- Defining a new essence type that requires a new data kind for segments
- Defining a new essence type that requires a new kind of Mob Slot and a new set of classes for Composition Metadata Objects

This section contains a brief description of how to define a new essence type that uses an existing data kind. Describing the requirements of defining a new data kind, a new kind of Mob Slot, or new classes for Composition Metadata Objects is beyond the scope of this document.

To define a new essence type, you must:

- Define a new subclass of FileDescriptor or a new subclass of EssenceDescriptor for Source Metadata Objects that are not File Source Metadata Objects
- Define a new subclass of EssenceData or use an existing class
- Create a plug-in essence codec that can import and export the essence data based on the information in the File Descriptor

Typically, when defining a new essence format you can use the existing classes for the Mob Slots and Segments in the Source Metadata Object, but you do have to define a new Essence Descriptor.

If the new essence type consists of a single kind of essence data, such as a single video stream or a static image, the Source Metadata Object should have a single Mob Slot. If the essence type is a compound format that has multiple tracks of essence data, the File Source Metadata Object should have a separate Mob Slot for each separate track of essence data.

.

6. Tips and Troubleshooting

Tips

Troubleshooting

Problem: Program creating new AAF file completes successfully, but AAF file is empty

Check that you have called the IAAFFile->Save method before calling IAAFFile->Close

Problem: Program modifying existing AAF file completes successfully, but AAF file is unchanged.

Check that you have called the IAAFFile->Save method before calling IAAFFile->Close

Problem: Program creating new AAF file or modifying existing AAF file completes successfully, but some new objects are missing from the file.

Objects are written to the AAF file only if they are attached to it. For example, the following code creates a CompositionMob:

```
check(pDictionary->CreateInstance( &AUID_AAFCompositionMob,
          IID_IAAFMob, (IUnknown **)&pCompMob));
```

But it will not be written to the AAF file unless it is attached to it with the following code:

```
// Add it to the file
check(pHeader->AppendMob(pCompMob));
```

Problem: Error interchanging data between applications.

When you are exporting an AAF file from one application and importing it into another and encounter an error, it can be difficult discovering the source of the error and fixing it. This section provides some basic information on diagnosing interchange errors, but you should consult the documentation and technical support for the exporting and importing applications for more detailed information. The following are some potential causes of interchange errors:

- The AAF file was damaged when it was transferred from one system to another. One frequent error caused by transferring an AAF file is when a transfer mechanism treats an AAF file as a text file. An AAF file contains binary data and when it is transferred as a text file, the internal structure of the AAF file is damaged.

- Others to be supplied