Adding Intelligence to Media

XMP MEDIA PRODUCTION SDK

PROGRAMMER'S GUIDE

Adobe®

Extensible Metadata Platform (XMP) Media Production SDK, Programmer's Guide.

Contents

Preface **5**

About this document 5
- How this document is organized 5
- Conventions used in this document 5

1 XMP Media Production SDK overview **6**

2 XMP Media Production SDK layout **7**

3 XMP Media Production Data Model **8**

Source List 8

Output List 8

Video/Audio Track List 9

Shot List 10

Shot Source List 11

Frame List 11

Default namespace 12

4 Extension mechanism for storing custom metadata **14**

5 Basic hierarchical representation of content **15**

6 Setting up the XMP Media Production SDK **16**

Supported platforms 16

Prerequisites for building XMP Media Production SDK 16

Steps to build the XMP Media Production SDK 16
- Installing Cmake 16
- Steps to build the XMP Media Production SDK on Windows 17
- Steps to build the XMP Media Production SDK on Macintosh 18
- Steps to build the XMP Media Production SDK on Linux 19

7 Code snippets for some common workflows **21**

Initializing XMP Media Production SDK 21

Terminating XMP Media Production SDK 21

Creating UMC object from a buffer 21

Creating an empty UMC object 21

Serializing a UMC to a buffer 21

Creating and adding sources to the UMC object 21

Creating and adding outputs 24
Creating and adding tracks within outputs 24
Creating and adding shots to tracks 25
Creating and adding ShotSources to shots 25
Creating and adding frames to shots 26
Creating and adding extensions to a UMC object 27

A References 34

Preface

About this document

This document, the *XMP Media Production SDK Programmer's Guide*, provides guidance to programmers who work with metadata using the XMP Media Production SDK. This XMP Media Production SDK is provided by Adobe® as a free Software Development Kit (SDK). This guide contains a lot of tutorial material for explaining the various concepts.

How this document is organized

This document has the following sections:

- Chapter 1, "XMP Media Production SDK overview", provides an introduction to the usage and purpose of XMP Media Production SDK.
- Chapter 2, "XMP Media Production SDK layout", describes the project structure of the XMP Media Production SDK.
- Chapter 3, "XMP Media Production Data Model", describes the data model of the XMP Media Production SDK and introduces the data structures and the concepts used.
- Chapter 4, "Extension mechanism for storing custom metadata", describes how custom data can be added in the XMP Media Production SDK.
- Chapter 5, "Basic hierarchical representation of content", pictorially represents the content hierarchy of the XMP Media Production SDK.
- Chapter 6, "Setting up the XMP Media Production SDK", describes the supported platforms for the SDK and the required dependencies. It also provides detailed steps on building the SDK on the supported platforms.
- Chapter 7, "Code snippets for some common workflows", provides code snippets for various common workflows and usage patterns anticipated in the XMP Media Production SDK. The code snippets are accompanied by some brief explanation and comments wherever deemed necessary.
- Chapter A, "References, describes the documents referred during the creation of the guide.

Conventions used in this document

The following type styles are used for specific types of text:

Typeface Style	Used for:
`Monospaced bold`	XMP property names. For example, `xmp:CreateDate`
`Monospaced Regular`	XML code and other literal values, such as value types and names in other languages or formats

1 XMP Media Production SDK overview

The *Adobe Extensible Metadata Platform* (XMP) is an international standard [ISO 16684-1]. The standardized data model and the associated operations are fully implemented as an XMP Toolkit Software Development Kit (SDK) by Adobe (http://www.adobe.com/devnet/xmp.html).

As part of enabling custom metadata for digital video production and distribution workflows via XMP, the XMP Media Production SDK is an extension layer over XMP Toolkit SDK. This extended layer allows convenient editing and storing of static, temporal and other custom metadata. The XMP Media Production SDK provides an extension mechanism for storing vendor specific metadata and other custom metadata. This process of adding metadata can be carried along with the composition throughout the video production and editing workflows.

2 XMP Media Production SDK layout

The following directories are present in the XMP Media Production SDK root directory:

Directory	Purpose/Contents
/	The root directory contains `CmakeLists.txt`, LICENSE file and Common Defines.cmake files. It contains build, public, private, imports and samples directories
`./build`	It contains Shared Config files for different platforms and all the build solutions/projects are generated here.
`./samples`	It contains the samples for XMP Media Production SDK and the Cmake scripts to build samples.
`./public/include`	It contains the public headers of the XMP Media Production SDK.
`./public/source`	It contains the public sources of the XMP Media Production SDK.
`./private`	It contains the include and source directories and contains the Cmake scripts to build the XMP Media Production library.
`./private/include`	It contains the private headers of the XMP Media Production SDK
`./private/source`	It contains the private source files of the XMP Media Production SDK
`./imports`	It contains the XMPToolkit SDK which is the required dependency for XMP Media Production SDK.

3 XMP Media Production Data Model

The XMP Media Production SDK uses simple properties, array properties, and structure properties as described in the XMP Specification. The *XMP Specification Part 1, Data Model, Serialization, and Core Properties* provides a complete overview of the XMP storage model, in addition to describing how XMP is serialized and discusses issues with RDF.

The XMP Media Production SDK data model uses the following concepts for the current and anticipated future use cases.

Source List

All media contents are composed from individual sources like images, audio, videos, and video frames. `SourceList` structure is equivalent to the concept of metadata of `bins` in digital video production and editing systems. In modern digital systems, a location for storing and organizing clips in the video project is referred to as a *bin*. The Source List structure can be represented as an unordered array of source or master clips numbering from 1 to N. The ordering of sources in this array is insignificant for video production and editing workflows.

```
Source List

    Source [1]

    Source [2]

    ...

    Source [N]
```

Each `Source` 1 through N, can contain the following list of metadata entries.

Source properties:

- Source Type - denotes the type of the source (Video | Audio | Image | VideoFrame)
- Clip Name - a string to identify the source or master clip
- InCount - denotes the offset in `EditUnits` from the start of the source
- Duration - duration of source clip in edit units
- TimeCode - time code object denoting the `timecode` of the source
- Edit Rate - `EditRate` object denoting the edit rate of the source. Edit Rate for audio sources is referred as the Audio Edit Rate
- Unique ID - a string which serves as an identification tag for the source unique to a project specific scope

Output List

A composition is referred to as an Output in the XMP Media Production SDK. It consists of audio, images, video and other content. Most of the necessary data is tightly coupled with the video and audio

components. Therefore, an XMP Media Production SDK Output is structured into a video and an audio section. Other data like subtitles, titles, captions, and edit rate are typically generated along with the composition throughout the video distribution workflows.

XMP Media Production SDK allows multiple outputs. For example, different edited compositions of the same title can be grouped into the same project within a single object. The top level structure of Outputs is parallel to the Source List and is referred to as the Output List. The XMP Media Production SDK Output List has a similar data structure as the Source List, i.e. an unordered array of Output data structures because ordering of outputs is insignificant in most video editing and production workflows.

```
Output List

    Output [1]

    Output [2]

    …

    Output [N]
```

The Output structure contain information specific to the composition with more specific entries organized under lower level structures. The current design includes properties like the Output Title, Output Unique ID, Number of Video and Audio Tracks, a Video Track List, an Audio Track List, Canvas Aspect Ratio and Image Aspect Ratio.

Output properties:

- Output Name - a string to identify the composition
- Title - a string for specifying the title of the composition
- Image Aspect Ratio - aspect ratio of the image action area not including the formatting black bars
- Canvas Aspect Ratio - aspect ratio of the video picture area from edge to edge including formatting black bars
- Video TrackList - a data structure containing an ordered list of the video tracks in the outputs
- Audio TrackList - a data structure containing an ordered list of the audio tracks in the outputs
- Number of Video Tracks
- Number of Audio Tracks
- Unique ID - a string which serves as an identification tag for the output, which is unique within a project scope

Video/Audio Track List

An output or a composition can have audio and/or video subparts which are represented by an ordered array called the Audio Track List and the Video Track List respectively.

```
Audio TrackList

    AudioTrack [1]

    AudioTrack [2]

    …
```

```
AudioTrack [N]


Video TrackList
    VideoTrack [1]
    VideoTrack [2]
    …
    VideoTrack [N]
```

The track structure includes all track specific information on the overall video or audio track such as name of the track, unique ID, and edit rate. Tracks also contain a Shot List which contains all the shots included in the track.

Tracks are represented using an ordered array.

Track properties:

- Track Name - a string to identify the name of the video or audio track
- Edit Rate - an Edit Rate object denoting the edit rate of the track
- Shot List - an ordered list of shots contained inside the track
- Unique ID - a string which serves as an identification tag for the track, which is unique within a project scope

Shot List

`ShotList` is an ordered list of shots present inside a track. The unit of media that is assembled into a composition is the shot or cut. A shot is a segment of video or audio, from a source clip in the Source List structure at the top level.

```
ShotList
    Shot [1]
    Shot [2]
    …
    Shot [N]
```

The complete Shot data structure consists of a unique ID, a shot type (currently shot type can be Clip or Transition), a Shot Source List, and a Frame List structure as explained below.

Shot properties:

- Shot Type - a string of value "Clip" or "Transition"
- ShotSourceList - an ordered list of shots contained inside the track
- FrameList - an ordered list of shots contained inside the track
- Unique ID - string which serves as an identification tag for the shot unique to a project scope

Shot Source List

A shot contains a Shot Source List which is an unordered array of Shot Source objects.

A shot source is used to describe effects like fade transitions and dissolve transitions. Fade-out transitions happen when the picture is gradually replaced by black screen or any other solid color. Fade-ins are the opposite: a solid color gradually gives way to picture. Dissolves happen when one shot is gradually replaces by the next. One shot disappears as the following shot appears and for a few seconds, they overlap, and both are visible.

```
ShotSourceList

    Shot Source [1]

    Shot Source [2]

    …

    Shot Source [N]
```

An individual shot source can contain the unique id, source in count, shot in count and the source.

Shot Source properties:

- Source InCount - denotes the offset in EditUnits with respect to the source of the shot source
- Shot InCount - denotes the offset in EditUnits with respect to the shot of the shot source
- Source - denotes the source from which the shot source has been taken
- Unique ID - a string which serves as an identification tag for the shot source unique to a project scope

Frame List

The individual frames in a shot are stored as a Frame List and it is represented as an ordered array of Frame objects.

```
FrameList

    Frame [1]

    Frame [2]

    …

    Frame [N]
```

Shots can contain metadata at the individual frame level. A frame structure is used for this purpose and it contains unique id, source, source in count and shot in count data fields.

Frame Properties

- Source InCount - denotes the offset in Edit Units with respect to the source of the frame
- Shot InCount - denotes the offset in Edit Units with respect to the shot of the frame
- Source - denotes the source from which the frame has been taken
- Unique ID - a string which serves as an identification tag for the frame, which is unique within a project scope

Default namespace

UMC (*Universal Metadata Container*) namespace is the default namespace for all XMP Media Production SDK concepts.

- The namespace URI shall be *http://ns.umc.com/xmp/1.0/UniversalMetadataContainer/*.
- The preferred namespace prefix is **umc**.

Here is a representation of a sample Universal Metadata Container object which contains sources and outputs:

```
UMC:SourceList (isArray)
    [1] UMC:Source (isStruct)
       UMC:EditRate="50/1"
       UMC:InCount="0"
       UMC:Duration="100"
       UMC:SourceType="Video"
       UMC:UniqueID="1"
       UMC:ClipName="Video Source Name"
     [2] UMC:Source (isStruct)
       UMC:SourceType="Audio"
       UMC:UniqueID="2"
       UMC:ClipName="Audio Source Name"
       UMC:EditRate="50/1"
       UMC:InCount="0"
       UMC:Duration="100"


UMC:Outputs (isArray)
    [1] UMC:Output (isStruct)
       UMC:OutputName = "example output for digital cinema"
       UMC:UniqueID = "3"
       UMC:Title = "example output"
       UMC:NumberVideoTrack = "1"
       UMC:NumberAudioTrack = "1"
       UMC:CanvasAspectRatio = "1:1"
       UMC:ImageAspectRatio = "1:1"
       UMC:AudioTrackList (isArray)
           [1] UMC:Track (isStruct)
```

```
        UMC:TrackName = "primary audio track"
        UMC:UniqueID = "4"
        UMC:EditRate = "24/1"

UMC:VideoTrackList (isArray)
    [1] UMC:Track (isStruct)
        UMC:TrackName = "primary video track"
        UMC:UniqueID = "5"
        UMC:EditRate = "50/1"
        UMC:ShotList (isArray)
            UMC:Shot (isStruct)
            UMC:UniqueID = "6"
            UMC:ShotType = "Clip"
            UMC:ShotSourceList (isArray)
                [1] UMC:ShotSource (isStruct)
                    UMC:InCount = "0"
                    UMC:ParentID = "1"
                    UMC:FrameList (isArray)
                        [1] UMC:Frame (isStruct)
                            UMC:UniqueID = "7"
                            UMC:SourceInCount = "0"
                        [2] UMC:Frame (isStruct)
                            UMC:UniqueID = "8"
                            UMC:SourceInCount = "1"
                        [3] UMC:Frame (isStruct)
                            UMC:UniqueID = "9"
                            UMC:SourceInCount = "2"
```

4 Extension mechanism for storing custom metadata

XMP Media Production SDK also provides an extension mechanism for storing and modifying custom metadata along with sources, outputs, tracks, frames, shots and shot sources. It is often required in video production workflows to store custom metadata at various junctures - at the source, at the composition level, at the shot level etc. Some examples for use cases of custom metadata are described below:

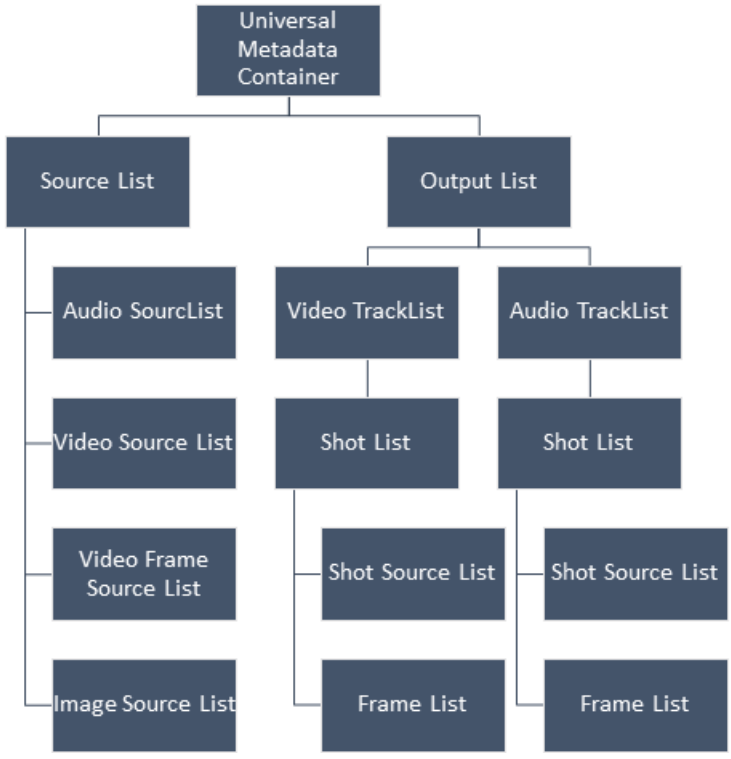
- A video editor might want to annotate a composition such that each shot has its own annotations. In this case, it is convenient to store custom metadata at the shot level.
- Mastering display (i.e. the display that is used to produce the archival version of a title) specifications are usually specified for each composed output or video track. These display properties should be ideally stored as custom metadata at the output or the video track level.

5 Basic hierarchical representation of content

The basic content hierarchy is given below:

```
Source List ( 1..N source files containing  images/videos/audios/ video frame sources)


Output List ( 1..N timelines of audio and video content) Props: ID, Title, Video/Audio
tracks,

    Track List ( 1..N list of tracks containing audio tracks/video tracks)


        Props: ID, Edit Rate, Name

        Shot List (1..N list of shots containing Clip shots/ Transition Shots)


            Props: ID, Duration

            FrameList (1..N list of frames)

            Shot SourceList ( 1..N list of shot sources created using existing sources)
```

The content hierarchy expressed in the form of a diagram:



NOTE: The main interface of the XMP Media Production SDK is referred to as the Universal Metadata Container (UMC).

6 Setting up the XMP Media Production SDK

The root directory of the XMP Media Production SDK is referred as the `XMP_MP_ROOT_DIR`.

Supported platforms

The XMP Media Production SDK is officially supported on the following platforms:

- Windows (compiler used, Microsoft Visual Studio 14 2015)
- Macintosh (compiler used, XCode 7.2)
- Linux (RHEL 7 and compiler used gcc 4.8.2)

However, it is possible to build XMP Media Production SDK on other platforms as well by making some minor changes in the Cmake build scripts.

Prerequisites for building XMP Media Production SDK

In the top level directory of XMP Media Production SDK, there is an `imports` directory.

It should contain the latest Adobe XMP Toolkit SDK, which can be obtained from the XMP Devnet page:

(http://www.adobe.com/devnet/xmp.html)

The XMP Toolkit SDK root directory should be copied into the `imports` directory of the XMP Media Production SDK such that the `imports` should contain the following path.

```
XMP_MP_ROOT_DIR/imports/XMPToolkit/
```

The XMP Toolkit SDK binaries should be built following the steps described in the *XMP Toolkit SDK Programmer's Guide*.

Steps to build the XMP Media Production SDK

The root directory of the XMP Media Production SDK is referred as the `XMP_MP_ROOT_DIR`.

Installing Cmake

NOTE: Cmake is required to build the SDK. You must have Cmake version 3.5.2 or above to build the SDK.

Obtain a copy of the CMake distribution for your platform from the following location:

http://www.cmake.org/cmake/resources/software.html

Cmake-gui tools comes bundled with Cmake for Macintosh and Windows platforms.

For Linux platforms, Cmake-gui can be conveniently installed by using the following command:

```
yum install cmake-gui
```

The steps given in this guide use the Cmake gui tool as this is the most convenient way to build XMP Media Production SDK.

Steps to build the XMP Media Production SDK on Windows

1. Extract the latest XMP Toolkit SDK into the `XMP_MP_ROOT_DIR` and build the XMP Toolkit SDK 64-bit dynamic binaries for Windows by using the steps mentioned in the *XMP Toolkit Programmer's Guide*.
2. Open the CMake GUI tool.
3. Set the path of the Source Code as the root directory of XMP Media Production SDK, which is `XMP_MP_ROOT_DIR`.
4. Set the path to build the binaries as `XMP_MP_ROOT_DIR/build/`.
5. Select the Visual Studio 14 2015 Win 64 generator to build the solution and click Generate.
6. The following projects are generated:
   - `UMCSDK.sln` is generated in the `XMP_MP_ROOT_DIR/build/` directory.
   - `Samples.sln` is generated in the `XMP_MP_ROOT/build/samples/` directory.
   - Individual sample solutions are generated as follows

     - `AddingSources.sln` is generated in the `XMP_MP_ROOT_DIR/build/samples/AddingSources/build/` directory.

     - `AddingExtensions.sln` is generated in the `XMP_MP_ROOT_DIR/build/samples/AddingExtensions/build/` directory.

     - `AddingOutputs.sln` is generated in the `XMP_MP_ROOT_DIR/build/samples/AddingOutputs/build/` directory.
7. Open the `UMCSDK.sln` in Visual Studio and select the `ALL_BUILD` target and build it. It will build all the UMC lib and the samples.
8. The debug binaries will be generated at `XMP_MP_ROOT_DIR/build/private/debug` and the release binaries will be generated at `XMP_MP_ROOT_DIR/build/private/`release.
9. The debug samples will be built as follows:
   - `AddingSources` executable is generated in the `XMP_MP_ROOT_DIR/build/samples/AddingSources/build/debug` directory.
   - `AddingExtensions` executable is generated in the `XMP_MP_ROOT_DIR/build/samples/AddingExtensions/build/debug` directory.
   - `AddingOutputs` executable is generated in the `XMP_MP_ROOT_DIR/build/samples/AddingOutputs/build/debug` directory.
10. The release samples will be built as follows:

- **AddingSources** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingSources/build/** directory.
- **AddingExtensions** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingExtensions/build/** directory.
- **AddingOutputs** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingOutputs/build/** directory.

Steps to build the XMP Media Production SDK on Macintosh

1. Extract the latest XMP Toolkit SDK into the **XMP_MP_ROOT_DIR** and build the XMP Toolkit SDK 64-bit frameworks for Macintosh using the steps mentioned in the *XMP Toolkit Programmer's Guide*.
2. Open the CMake GUI tool.
3. Set the path of the Source Code as the root directory of XMP Media Production SDK which is **XMP_MP_ROOT_DIR**.
4. Set the path to build the binaries as **XMP_MP_ROOT_DIR/build/**.
5. Select the XCode generator to build the xcode project and click Generate.
6. The following xcode projects are generated:
   - **UMCSDK.xcodeproj** is generated in the **XMP_MP_ROOT_DIR/build/** directory.
   - **Samples.xcodeproj** is generated in the **XMP_MP_ROOT/build/samples/** directory.
   - Individual sample solutions are generated as follows

     - **AddingSources.xcodeproj** is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingSources/build/** directory.

     - **AddingExtensions.xcodeproj** is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingExtensions/build/** directory.

     - **AddingOutputs.xcodeproj** is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingOutputs/build/** directory.
7. Open the **UMCSDK.xcodproj** in Visual Studio and select the **ALL_BUILD** target and build it. It will build all the UMC lib and the samples.
8. The debug binaries will be generated at **XMP_MP_ROOT_DIR/build/private/debug** and the release binaries will be generated at **XMP_MP_ROOT_DIR/build/private/**release.
9. The debug samples will be built as follows:
   - **AddingSources** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingSources/build/**debug directory.
   - **AddingExtensions** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingExtensions/build/**debug directory.
   - **AddingOutputs** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingOutputs/build/debug** directory.
10. The release samples will be built as follows:

- **AddingSources** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingSources/build/** directory.
- **AddingExtensions** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingExtensions/build/** directory.
- **AddingOutputs** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingOutputs/build/** directory.

Steps to build the XMP Media Production SDK on Linux

1. Extract the latest XMP Toolkit SDK into the **XMP_MP_ROOT_DIR** and build the XMP Toolkit SDK 64-bit Linux binaries using the steps mentioned in the *XMP Toolkit Programmer's Guide*.
2. Open the cmake-gui tool.
3. Set the **CMAKE_BUILD_TYPE** variable in cmake-gui as either "debug" or "release" depending on whether you want to build the debug binaries or the release binaries.
4. Set the path of the Source Code as the root directory of XMP Media Production SDK which is **XMP_MP_ROOT_DIR**.
5. Set the path to build the binaries as **XMP_MP_ROOT_DIR/build/**.
6. Select the Linux Makefile generator to build the project and click Generate.
7. The following Makefiles are generated
   - UMCSDK Makefile is generated in the **XMP_MP_ROOT_DIR/build/** directory.
   - Samples Makefile is generated in the **XMP_MP_ROOT/build/samples/** directory.
   - Individual sample solutions are generated as follows

     - **AddingSources** Makefile is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingSources/build/** directory.

     - **AddingExtensions** Makefile is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingExtensions/build/** directory.

     - **AddingOutputs** Makefile is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingOutputs/build/** directory.
8. To build the binaries or the executables run **make -k all** on the respective makefile.
9. The debug binaries will be generated at **XMP_MP_ROOT_DIR/build/private/debug** and the release binaries will be generated at **XMP_MP_ROOT_DIR/build/private/release**.
10. The debug samples will be built as follows:
    - **AddingSources** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingSources/build/debug** directory.
    - **AddingExtensions** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingExtensions/build/debug** directory.
    - **AddingOutputs** executable is generated in the **XMP_MP_ROOT_DIR/build/samples/AddingOutputs/build/debug** directory.

11. The release samples will be built as follows:

    - **`AddingSources`** executable is generated in the **`XMP_MP_ROOT_DIR/build/samples/AddingSources/build/`** directory.
    - **`AddingExtensions`** executable is generated in the **`XMP_MP_ROOT_DIR/build/samples/AddingExtensions/build/`** directory.
    - **`AddingOutputs`** executable is generated in the **`XMP_MP_ROOT_DIR/build/samples/AddingOutputs/build/`** directory.

7 Code snippets for some common workflows

Initializing XMP Media Production SDK

UMC should be initialized before using any of the UMC APIs.

```
UMC_Initialize();
```

Terminating XMP Media Production SDK

UMC should have a matching terminate call for each `UMC_Initialize()` call. Ideally, UMC should be initialized and terminated only once.

```
UMC_Terminate();
```

Creating UMC object from a buffer

UMC object can be created using a string buffer in the following manner. The buffer should be a valid serialization of the UMC object. A UMC object is the main container object which contains all the sources and all the outputs. UMC object also contains functions to serialize the UMC object and to register custom data handlers for handling custom data objects.

```
// the buffer is umcBuffer which is a std::string

auto sp = IUMC::CreateUMCFromBuffer(umcBuffer)
```

Creating an empty UMC object

An empty UMC object can be created using the `CreateEmptyUMC()` call.

```
spIUMC sp = IUMC::CreateEmptyUMC();
```

Serializing a UMC to a buffer

A UMC object can be serialized to a string using the `SerializeToBuffer()` call.

```
std::string buffer = umc->SerializeToBuffer()
```

Creating and adding sources to the UMC object

A UMC can contain various kinds of sources.

The sources are represented using the `ISource.h` interface. Source objects can be added to or deleted from a UMC object. Some source objects can be used to compose some output objects.

The specific types of valid sources can be:

- **Video Source**

Video sources are represented using the `IVideoSource.h` interface. A video source has Video Edit Rate, Audio Edit Rate, TimeCode, Incount and Duration properties which can be retrieved/modified.

- **Audio Source**

  Audio sources are represented using the `IAudioSource.h` interface. An audio source has Audio Edit Rate, TimeCode, Incount and Duration properties which can be retrieved/modified.

- **Video Frame Source**

  Video Frame sources are represented using the `IVideoFrameSource.h` interface. A video frame source has Video Source, TimeCode and Incount properties which can be retrieved/modified.

- **Image Source**

  Image sources are represented using the `IImageSource.h` interface

  Please refer to the `AddingSources` sample or the code snippet provided below for an illustration of using the various kinds of sources.

```
spIUMC sp = IUMC::CreateEmptyUMC();


    // add 5 video source ( 1 filled, 1 empty )
    auto videoSource = sp->AddVideoSource();
    videoSource->SetClipName( "clipNamev1" );
    videoSource->SetInCount( 5 );
    videoSource->SetDuration( 50 );
    videoSource->SetVideoEditRate( EditRate( 24000, 1001 ) );
    videoSource->SetAudioEditRate( 48000 );
    videoSource->SetTimeCode( TimeCode( TimeCode::k23_976Fps, "02:04:12:21" ) );


    sp->AddVideoSource();
    sp->AddVideoSource();
    sp->AddVideoSource();
    sp->AddVideoSource();


    // add 5 audio source( 2 filled, 1 blank )
    auto audioSource = sp->AddAudioSource();
    audioSource->SetClipName( "clipNamea1" );
    audioSource->SetInCount( 10 );
    audioSource->SetAudioEditRate( EditRate( 48000 ) );
    audioSource->SetTimeCode( TimeCode( TimeCode::k25Fps, 1, 2, 3, 4 ) );
```

```
    audioSource = sp->AddAudioSource();
    audioSource->SetClipName( "clipNamea2" );
    audioSource->SetDuration( 200 );
    audioSource->SetAudioEditRate( EditRate( 44100 ) );
    audioSource->SetTimeCode( TimeCode( TimeCode::k29_97Fps, 1, 2, 3, 4, true ) );

    sp->AddAudioSource();
    sp->AddAudioSource();
    sp->AddAudioSource();

    // add 5 images ( 1 filled, 2 blank )
    auto imageSource = sp->AddImageSource();
    imageSource->SetClipName( "clipNamei1" );

    sp->AddImageSource();
    sp->AddImageSource();
    sp->AddImageSource();
    sp->AddImageSource();

    // add 5 video frame sources
    auto videoFrameSource = sp->AddVideoFrameSource( videoSource );
    videoFrameSource->SetClipName( "clipNamevf1" );
    videoFrameSource->SetInCount( 5 );
    sp->AddVideoFrameSource( videoSource );
    sp->AddVideoFrameSource( videoSource );
    sp->AddVideoFrameSource( videoSource );
    sp->AddVideoFrameSource( videoSource );
        // Get all video sources
          auto videoSources = sp->GetAllVideoSources();
        // Iterate over all video sources
for( int i= 1; i<sp->VideoSourceCount(); i++)
{
         EditRate rate=videoSources[i]>GetVideoEditRate();
          auto inCount = videoSources[i]->GetInCount();
          auto duration = videoSources[i]->GetDuration();
```

```
}
```

Creating and adding outputs

Various kinds of outputs can be added to a UMC object. Outputs are represented using the `IOutput` interface. Outputs contain audio tracks, video tracks, output name, output title, image aspect ratio, canvas aspect ratio, audio edit rate, and video edit rate properties which can be added, deleted or modified.

For an illustration of adding outputs to a UMC object, please refer to the code snippet below or the `AddingOutputs` sample.

```
spIUMC sp = IUMC::CreateEmptyUMC();

    auto output1 = sp->AddOutput();

    output1->SetAudioEditRate( EditRate( 48000 ) );

    output1->SetVideoEditRate( EditRate( 24 ) );

    output1->SetName( "Output One" );

    output1->SetTitle( "Video for Output One" );

    output1->SetImageAspectRatio( AspectRatio( 1080, 720 ) );

    output1->SetCanvasAspectRatio( AspectRatio( 640, 480 ) );

    sp->AddOutput();


    // Getting outputs from UMC

    IUMC::OutputList outputs = sp->GetAllOutputs();
```

Creating and adding tracks within outputs

An Output object contains video and audio tracks. A track is represented using the `ITrack` interface and has clip shots, transition shots, and track name properties which can be added, modified or deleted.

1. An audio track is represented using the `IAudioTrack` interface. An audio track has Audio Edit rate property which can be retrieved or modified; in addition to properties of an `ITrack` object.
2. A video track is represented using the `IVideoTrack` interface. A video track has Video Edit Rate and Audio Edit Rate properties which can be retrieved or modified; in addition to properties of an `ITrack` object.

For an illustration of adding tracks to a UMC object, please refer to the code snippet below or the `AddingOutputs` sample.

```
spIUMC sp = IUMC::CreateEmptyUMC();

    auto output1 = sp->AddOutput();

    auto videoTrack1 = output1->AddVideoTrack();

    videoTrack1->SetName( "videoTrack1" );

    videoTrack1->SetVideoEditRate( EditRate( 24000, 1001 ) );

    videoTrack1->SetAudioEditRate( 48000 );
```

```
output1->AddVideoTrack();

output1->AddVideoTrack();


auto audioTrack1 = output1->AddAudioTrack();

audioTrack1->SetName( "audioTrack1" );

audioTrack1->SetAudioEditRate( 44000 );

auto audioTrack2 = output1->AddAudioTrack();

audioTrack2->SetName( "audioTrack2" );

output1->AddAudioTrack();

output1->AddAudioTrack();
```

Creating and adding shots to tracks

A shot object contains frames and shot sources. A shot is represented using the `IShot` interface and has Frames, ShotSources, InCount and Duration properties which can be added, modified or deleted. Shots are of two kinds:

1. A clip shot is represented using the `IClipShot` interface.
2. A transition shot is represented using the `ITransitionShot` interface.

For an illustration of adding shots to a UMC object, please refer to the code snippet below or the `AddingOutputs` sample.

```
spIUMC sp = IUMC::CreateEmptyUMC();

auto output1 = sp->AddOutput();

auto videoTrack1 = output1->AddVideoTrack();

auto clipShot1 = videoTrack1->AddClipShot();

clipShot1->SetInCount( 10 );

clipShot1->SetDuration( 15 );

videoTrack1->AddClipShot();

videoTrack1->AddClipShot();

auto transitionShot1 = videoTrack1->AddTransitionShot();

transitionShot1->SetInCount( 8 );

transitionShot1->SetDuration( 3 );

videoTrack1->AddTransitionShot();
```

Creating and adding ShotSources to shots

A shot source is represented using the `IShotSource` interface and has the Source, Source InCount, Source Duration and Shot InCount properties which can be added, modified or deleted. A shot source object is added to a shot object.

For an illustration of adding shot sources to a UMC object, please refer to the code snippet below or the `AddingOutputs` sample.

```
auto  sp = IUMC::CreateEmptyUMC();

    auto output1 = sp->AddOutput();

    auto source1 = sp->AddVideoSource();

    source1->SetClipName( "source 1" );

    auto source2 = sp->AddVideoSource();

    source2->SetClipName( "source 2" );

    auto videoTrack1 = output1->AddVideoTrack();


    auto clipShot1 = videoTrack1->AddClipShot();

    clipShot1->SetInCount( 10 );

    clipShot1->SetDuration( 15 );


    auto shotSource1 = clipShot1->AddShotSource( source1 );

    auto transitionShot1 = videoTrack1->AddTransitionShot();

    auto shotSource2 = transitionShot1->AddShotSource( source1 );

    auto shotSource3 = transitionShot1->AddShotSource( source2 );
```

Creating and adding frames to shots

A frame is represented using the `IFrame` interface and has the Source, Source InCount and the Shot In Count properties which can be added, modified or deleted.

For an illustration of adding frames to a UMC object, please refer to the code snippet below or the `AddingOutputs` sample.

```
spIUMC sp = IUMC::CreateEmptyUMC();

    auto output1 = sp->AddOutput();

    auto source1 = sp->AddVideoSource();

    source1->SetClipName( "source 1" );

    auto videoTrack1 = output1->AddVideoTrack();


    auto clipShot1 = videoTrack1->AddClipShot();

    clipShot1->SetInCount( 10 );

    clipShot1->SetDuration( 15 );


    auto frame1 = clipShot1->AddFrame( source1 );
```

```
frame1->SetShotInCount( 12 );


auto frame2 = clipShot1->AddFrame( source1 );
frame2->SetShotInCount( 13 );


auto frame3 = clipShot1->AddFrame( source1 );
frame3->SetShotInCount( 14 );
```

Creating and adding extensions to a UMC object

An extension or custom data can be added to any UMC object or a UMC source object or a UMC output object. Extensions should be added to a particular object for storing metadata about that object which can't be specified by the default properties of that object.

An extension is created by extending the `ICustomData` interface and implementations for the functions present in the `ICustomData` interface. Empty implementations can be provided for the functions which are not being used. Additional properties and functions can be added to the extension for storing or modifying custom metadata.

In order to serialize and parse an extension, it is required to create an extension handler for every extension. The extension handler also needs to be registered by invoking the `RegisterCustomDataHandler()` function of the IUMC interface. The extension handler's purpose is to define how the extension is going to be serialized and how different properties are going to be serialized.

An extension handler is created by implementing the `ICustomDataHandler` interface. In addition to the `ICustomDataHandler` functions, additional properties and functions can be defined in the extension handler.

`Serialize()` function should be implemented to define how the extension is going to be serialized as a whole during UMC serialization.

`AddKeyValuePair()` function should be implemented in case the extension wants to add some simple properties or key-value pairs.

`BeginStructure()` and `EndStructure()` functions should be implemented in case the extension wants to create structure properties.

`BeginArray()` and `EndArray`() functions should be implemented in case the extension wants to create lists or arrays of homogeneous properties.

For an illustration of adding extensions to a UMC object, please refer to the code snippet below or the `AddingExtensions` sample.

```
//First create an extension class extending from the UMC::ICustomData class.


#include "interfaces/ICustomData.h"
#include "interfaces/IUMCNode.h"


namespace DisplayNamespace {
```

```
    using namespace UMC;

    class Display
        : public UMC::ICustomData
    {
    public:
        Display();
        std::string getID();
        void setID( const std::string & id );
        std::string getName();
        void setName( const std::string & n );
        double getDiagonalSize();
        void setDiagonalSize( const double & dz );

        virtual void SetParentNode( const wpIUMCNode & parentNode );

        virtual wpcIUMCNode GetParentNode() const;
        virtual wpIUMCNode GetParentNode();

        virtual ~Display();

        static const std::string & GetNameSpace();
        static const std::string & GetPropertyName();

    protected:
        std::string_id;
        std::string_name;
        double    _diagonalSize;
        wpIUMCNode_parent;
    };
}

namespace  DisplayNamespace {
```

```
void Display::SetParentNode( const wpIUMCNode & parentNode ) {
    _parent = parentNode;
}

wpcIUMCNode Display::GetParentNode() const {
    return _parent;
}

wpIUMCNode Display::GetParentNode() {
    return _parent;
}

std::string Display::getID() {
    return _id;
}

void Display::setID( const std::string & id ) {
    _id = id;
}

std::string Display::getName() {
    return _name;
}

void Display::setName( const std::string & n ) {
    _name = n;
}

double Display::getDiagonalSize() {
    return _diagonalSize;
}

void Display::setDiagonalSize( const double & dz ) {
    _diagonalSize = dz;
}
```

```
    Display::Display()
        : _diagonalSize( 32.0 ) { }

    Display::~Display() {

    }

    const std::string & Display::GetNameSpace() {
        static std::string nameSpace( "www.customnamespace.com/1.0/" );
        return nameSpace;
    }

    const std::string & Display::GetPropertyName() {
        static std::string name( "display" );
        return name;
    }

}
```

Then create an extension handler extending from the `UMC::IExtensionHandler` class.

```
//Create the custom handler for the above extension by extending the ICustomDataHandler
interface.
#include "interfaces/ICustomDataHandler.h"
#include "DisplayExtension.h"

namespace DisplayNamespace {
    using namespace UMC;

    class DisplayHandler
        : public ICustomDataHandler
    {
    public:
        virtual bool BeginCustomData();
        virtual spICustomData EndCustomData();
```

```
        virtual bool BeginStructure( const std::string & structureName );
        virtual bool EndStructure( const std::string & structureName );

        virtual bool BeginArray( const std::string & arrayName );
        virtual bool EndArray( const std::string & arrayName );

        virtual bool AddKeyValuePair( const std::string & key, const std::string & value
);

        virtual bool Serialize( const spICustomData & data, const spICustomDataHandler &
customDataCreator ) const;

    protected:
        shared_ptr< Display > _display;

    };
}

#include "DisplayExtensionHandler.h"

namespace DisplayNamespace {

    bool DisplayHandler::BeginCustomData() {
        _display = std::make_shared< Display >();
        return true;
    }

    spICustomData DisplayHandler::EndCustomData() {
        auto it = _display;
        _display.reset();
        return it;
    }

    bool DisplayHandler::BeginStructure( const std::string & structureName ) {
        return false;
```

```
    }

    bool DisplayHandler::EndStructure( const std::string & structureName ) {
        return false;
    }

    bool DisplayHandler::BeginArray( const std::string & arrayName ) {
        return false;
    }

    bool DisplayHandler::EndArray( const std::string & arrayName ) {
        return false;
    }

    bool DisplayHandler::AddKeyValuePair( const std::string & key, const std::string &
value ) {
        if ( key.compare( "id" ) == 0 ) {
            _display->setID( value );
            return true;
        } else if ( key.compare( "name" ) == 0 ) {
            _display->setName( value );
            return true;
        } else if ( key.compare( "diagonalSize" ) == 0 ) {
            _display->setDiagonalSize( std::stod( value ) );
            return true;
        }
        return false;
    }

    bool DisplayHandler::Serialize( const spICustomData & data, const
spICustomDataHandler & customDataCreator ) const {
        shared_ptr< Display > display = dynamic_pointer_cast< Display >( data );
        if ( display ) {
            if ( !customDataCreator->BeginCustomData() ) return false;
            if ( !display->getName().empty() )
                customDataCreator->AddKeyValuePair( "name", display->getName() );
```

```
            if ( !display->getID().empty() )

                customDataCreator->AddKeyValuePair( "id", display->getID() );

            if ( display->getDiagonalSize() != 0.0 )

                customDataCreator->AddKeyValuePair( "diagonalSize", std::to_string(
(long double )display->getDiagonalSize() ) );

            customDataCreator->EndCustomData();

            return true;

        }

        return false;

    }


}

 // Create an object of the extension and add it to the object

            auto umc = UMC::IUMC::CreateEmptyUMC();

    std::shared_ptr< DisplayNamespace::Display > display = std::make_shared<
DisplayNamespace::Display >();

    display->setID( "id-1");

    display->setName( "reference display - 1" );

    display->setDiagonalSize( 52.0 );

 // Adding the custom data

    umc->SetCustomData( DoVi::Display::GetNameSpace(),
DisplayNamespace::Display::GetPropertyName(), display );

//Register the extension data handler

UMC::IUMC::RegisterCustomNodeHandler( DoVi::Display::GetNameSpace(),
DisplayNamespace::Display::GetPropertyName(),

        std::make_shared< DisplayNamespace::DisplayHandler >() );

// Getting the extension from UMC

std::shared_ptr< DisplayNamespace::Display >parsedDisplay =
std::dynamic_pointer_cast< DisplayNamespace::Display >(

        um1->GetCustomData( DisplayNamespace::Display::GetNameSpace(),
DisplayNamespace::Display::GetPropertyName() ) );
```

A References

- http://www.adobe.com/devnet/xmp.html
- The *XMP Specification Part 1, Data Model, Serialization, and Core Properties*.
- The XMP Toolkit SDK Programmer's Guide.
- The XMP Toolkit SDK Addendum for the Programmers' Guide.