



AMD AMF Reference Manual

August 2014

© 2014 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, AMD Accelerated Parallel Processing, the AMD Accelerated Parallel Processing logo, ATI, the ATI logo, Radeon, FireStream, FirePro, Catalyst, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft, Visual Studio, Windows, and Windows Vista are registered trademarks of Microsoft Corporation in the U.S. and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. The information contained herein may be of a preliminary or advance nature and is subject to change without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.



Advanced Micro Devices, Inc.

One AMD Place

P.O. Box 3453

Sunnyvale, CA 94088-3453

www.amd.com

Contents

Chapter 1 Overview

Chapter 2 AMF API

Elementary Data Type	2-1
C++ Interfaces	2-2
AMF Interface	2-2
AMFInterface::Acquire()	2-2
AMFInterface::Release()	2-2
AMFInterface::QueryInterface()	2-2
AMFPropertyStorage	2-3
AMFPropertyStorage::SetProperty()	2-3
AMFPropertyStorage::SetProperty()	2-3
AMFPropertyStorage::GetProperty	2-4
AMFPropertyStorage::GetProperty	2-4
AMFPropertyStorage::GetPropertyString	2-5
AMFPropertyStorage::GetPropertyWString	2-5
AMFPropertyStorage::HasProperty()	2-6
AMFPropertyStorage::GetPropertyCount()	2-6
AMFPropertyStorage::GetPropertyAt()	2-6
AMFPropertyStorage::Clear()	2-6
AMFPropertyStorage::AddTo()	2-7
AMFPropertyStorage::CopyTo()	2-7
AMFPropertyStorage::AddObserver()	2-7
AMFPropertyStorage::RemoveObserver()	2-8
AMFPropertyStorageEx	2-8
AMFPropertyStorageEx::GetPropertiesInfoCount()	2-8
AMFPropertyStorageEx::GetPropertyInfo()	2-9
AMFPropertyStorageEx::GetPropertyInfo()	2-9
AMFPropertyStorageEx::ValidateProperty()	2-9
2.1 Memory	2-10
AMFData	2-10
AMFData::GetMemoryType()	2-10
AMFData::Duplicate()	2-11
AMFData::Convert()	2-11
AMFData::GetDataType()	2-11
AMFData::IsReusable()	2-11
AMFBuffer	2-12
AMFData::SetPts()	2-12
AMFData::GetPts()	2-12
AMFData::SetDuration()	2-12
AMFData::GetDuration()	2-12

AMFBuffer::GetSize()	2-13
AMFBuffer::GetNative().....	2-13
AMFBuffer::AddObserver().....	2-13
AMFBuffer::RemoveObserver().....	2-13
AMFBuffer::AddObserver().....	2-14
AMFBuffer::RemoveObserver().....	2-14
AMFSurface.....	2-14
AMFSurface::GetFormat().....	2-15
AMFSurface::GetPlanesCount().....	2-15
AMFSurface::GetPlaneAt()	2-15
AMFSurface::GetPlane()	2-15
AMFSurface:: GetFrameType().....	2-15
AMFSurface::RemoveObserver()	2-16
AMFSurface::AddObserver()	2-16
AMFSurface:: SetFrameType()	2-16
AMFSurface::AddObserver()	2-16
AMFSurface::RemoveObserver()	2-17
AMFSurface::SetCrop().....	2-17
AMFPlane	2-17
AMFPlane::GetType()	2-17
AMFPlane::GetNative().....	2-18
AMFPlane:: GetPixelSizelnBytes().....	2-18
AMFPlane::GetOffsetX().....	2-18
AMFPlane::GetOffsetY().....	2-18
AMFPlane::GetWidth()	2-18
AMFPlane:: GetHPitch()	2-19
AMFPlane:: GetVPitch()	2-19
AMFContext	2-19
AMFPlane::GetHeight()	2-19
AMFContext:: Terminate()	2-20
AMFContext:: InitDX9()	2-20
AMFContext:: GetDX9Device ().....	2-20
AMFContext:: LockDX9 ()	2-20
AMFContext:: UnlockDX9 ().....	2-20
AMFContext:: InitDX11()	2-21
AMFContext:: GetDX11Device ().....	2-21
AMFContext:: LockDX11 ()	2-21
AMFContext:: UnlockDX11 ().....	2-21
AMFContext:: InitOpenCL ()	2-22
AMFContext:: GetOpenCLContext ()	2-22
AMFContext:: GetOpenCLCommandQueue().....	2-22
AMFContext:: GetOpenCLDeviceID ()	2-22
AMFContext:: LockOpenCL ()	2-22
AMFContext:: UnlockOpenCL ().....	2-23
AMFContext:: InitOpenGL ().....	2-23
AMFContext:: GetOpenGLContext ()	2-23
AMFContext:: GetOpenGLDrawable ()	2-23
AMFContext:: LockOpenGL ()	2-24
AMFContext:: UnlockOpenGL ()	2-24
AMFContext:: AllocBuffer()	2-24
AMFContext:: CreateBufferFromHostNative ()	2-25
AMFContext:: AllocSurface().....	2-25
AMFContext:: CreateSurfaceFromHostNative ()	2-26

AMFContext:: CreateSurfaceFromDX9Native ()	2-26
AMFContext:: CreateSurfaceFromDX11Native ()	2-27
AMFContext:: CreateSurfaceFromOpenCLNative()	2-27
AMFComponent	2-28
AMFComponent:: Init()	2-28
AMFComponent:: Relnit ()	2-28
AMFComponent:: Drain ()	2-29
AMFComponent:: Flush ()	2-29
AMFComponent:: SubmitInput ()	2-29
AMFComponent:: QueryOutput ()	2-29
AMFComponent:: Terminate ()	2-29
AMFDataAllocatorCB	2-30
AMFComponent:: GetContext ()	2-30
AMFComponent:: SetOutputDataAllocatorCB ()	2-30
AMFDataAllocatorCB : AllocBuffer	2-30
AMFDataAllocatorCB : AllocSurface	2-31
AMFCaps	2-31
AMFCaps: GetAccelerationType	2-31
AMFCaps: GetInputCaps	2-32
AMFCaps: GetOutputCaps	2-32
AMFIOCaps	2-32
AMFIOCaps: GetWidthRange	2-33
AMFIOCaps: GetHeightRange	2-33
AMFIOCaps: GetVertAlign	2-33
AMFIOCaps: GetNumOfFormats	2-33
AMFIOCaps: GetFormatAt	2-34
AMFIOCaps: GetNumOfMemoryTypes	2-34
AMFIOCaps: GetMemoryTypeAt	2-35
AMFIOCaps: IsInterlacedSupported	2-35
AMFDecoderCaps	2-35
AMFDecoderCaps: GetMaxNumOfStreams	2-36
AMFEncoderCaps	2-36
AMFEncoderCaps: GetMaxBitrate	2-36
AMFEncoderCaps: GetMaxNumOfStreams	2-36
AMFH264EncoderCaps	2-37
AMFH264EncoderCaps: GetNumOfSupportedProfiles	2-37
AMFH264EncoderCaps: GetProfile	2-37
AMFH264EncoderCaps: GetNumOfSupportedLevels	2-37
AMFH264EncoderCaps: GetLevel	2-38
AMFH264EncoderCaps: GetNumOfRateControlMethods	2-38
AMFH264EncoderCaps: GetRateControlMethod	2-38
AMFH264EncoderCaps: GetMaxSupportedJobPriority	2-39
AMFH264EncoderCaps: IsBPictureSupported	2-39
AMFH264EncoderCaps: GetNumOfReferenceFrames	2-39
AMFH264EncoderCaps: CanOutput3D	2-40
AMFH264EncoderCaps: GetMaxNumOfTemporalLayers	2-40
AMFTraceWriter	2-40
AMFTraceWriter: Write	2-40
AMFTraceWriter: Flush	2-41
AMFTraceSetPath	2-41
AMFTraceGetPath	2-41
AMFTraceEnableWriter	2-42
AMFTraceWriterEnabled	2-42

AMFTraceSetGlobalLevel	2-42
AMFTraceGetGlobalLevel	2-42
AMFTraceSetWriterLevel	2-42
AMFTraceGetWriterLevel	2-43
AMFTraceSetWriterLevelForScope	2-43
AMFTraceGetWriterLevelForScope	2-43
AMFTraceRegisterWriter	2-43
AMFTraceUnregisterWriter	2-44
AMFEnablePerformanceMonitor	2-44
AMFPerformanceMonitorEnabled	2-44
AMFAssertsEnable	2-44
AMFAssertsEnabled	2-44

**Chapter 3
Enumerations**

Memory-related enumerations	3-1
AMF_MEMORY_TYPE	3-1
AMF_DATA_TYPE	3-1
AMF_SURFACE_FORMAT	3-2
AMF_PLANE_TYPE	3-2
AMF_FRAME_TYPE	3-3
.....	3-6
Result-related enumerations	3-6
AMF_RESULT	3-6
.....	3-8
Video Encoder enumerations	3-8
AMF_VIDEO_ENCODER_USAGE_ENUM	3-8
AMF_VIDEO_ENCODER_PROFILE_ENUM	3-8
AMF_VIDEO_ENCODER_SCANTYPE_ENUM	3-9
AMF_VIDEO_ENCODER_RATE_CONTROL_METHOD_ENUM	3-9
AMF_VIDEO_ENCODER_QUALITY_PRESET_ENUM	3-9
AMF_VIDEO_ENCODER_PICTURE_STRUCTURE_ENUM	3-9
AMF_VIDEO_ENCODER_PICTURE_TYPE_ENUM	3-10
AMF_VIDEO_ENCODER_OUTPUT_DATA_TYPE_ENUM	3-10
.....	3-11
Video Decoder enumerations	3-11
AMF_VIDEO_DECODER_MODE_ENUM	3-11
AMF_TIMESTAMP_MODE_ENUM	3-11
.....	3-11
Video Converter enumerations	3-11
AMF_VIDEO_CONVERTER_SCALE_ENUM	3-11
AMF_VIDEO_CONVERTER_COLOR_PROFILE_ENUM	3-12
.....	3-12
Acceleration enumerations	3-12
AMF_ACCELERATION_TYPE	3-12
.....	3-13
Job Priority enumerations	3-13
H264EncoderJobPriority	3-13
.....	3-13
Property Access enumerations	3-13
AMF_PROPERTY_ACCESS_TYPE	3-13
.....	3-14
Variant-specific enumerations	3-14

AMF_VARIANT_TYPE	3-14
Chapter 4	
Structure Definitions	
AMFRect	4-1
AMFSize	4-1
AMFRatio	4-2
AMFPoint	4-2
AMFRate	4-2
AMFEnumDescriptionEntry	4-3
AMFColor	4-3
AMFPropertyInfo	4-4
AMFVariantStruct	4-6
Chapter 5	
Using the AMD AMF API	
5.1 Device Selection (DirectX9/Direct11/OpenGL/OpenCL)	5-3
5.2 Video Encoder I/O	5-4
5.2.1 Video Encoder Input	5-4
5.2.2 Video Encoder Output	5-4
Chapter 6	
Pipeline Framework	
6.1 Framework	6-1
6.2 Class definitions of pipeline elements	6-2
Pipeline Element	6-2
Pipeline	6-3
6.3 Application Workflow	6-4
6.3.1 Initialize the pipeline	6-4
PipelineConnector	6-9
Slot	6-10
InputSlot	6-10
OutputSlot	6-11
6.3.2 Start/Run the pipeline	6-12
6.3.3 Terminate the pipeline	6-12

Figures

5.1	Typical workflow for an application using the AMD AMF API.....	5-2
5.2	AMF Interfaces used in the AMD Video Encoder.....	5-3
5.3	Retrieving the encoded video bitstream buffers.....	5-4
6.1	Pipeline Framework and Elements.....	6-1

Preface

About This Document

This document describes the AMD Media Framework (AMF) for real-time processing of multimedia. It also describes how to use the AMF to implement multimedia use-cases.

Audience

This document is intended for media processing developers, multimedia engineering managers, and multimedia architects making use of the AMD media processing technology.

Organization

This document begins with an overview of the AMD AMF API Framework. [Chapter 2](#) lists and describes the APIs used in AMF. [Chapter 3](#) lists and defines the enumerations. [Chapter 4](#) lists and describes the structure definitions. [Chapter 5](#) describes how to use the AMD AMF API. [Chapter 6](#) describes the pipeline framework.

Conventions

The following definitions, acronyms, and abbreviations are used in this document.

Term	Definition	Comments
Stream SDK	Accelerated Parallel Processing	AMD SDK implementing OpenCL spec
OCL	OpenCL	AMD SDK implementing OpenCL spec
MF	Media Foundation	Current video/audio framework in Windows
MFT	Media Foundation Transform	Main element of Media Foundation (filter)
MMD	Multi Media Driver	AMD driver for low-level multimedia functionality
UVD	Unified Video Decoder	Fixed function video decoder hardware
VCE	Video Compression Engine	Fixed function H.264 video encoder hardware
AMF	AMD Media Framework	Internal AMD C++ SDK created to build flexible pipelines
EG	Evergreen	GPU family
SI	Southern Islands	GPU family
DEM	Display Encode Mode	Direct connection of encoder and display – VCE mode

WinRT		Short name for Windows Store Application API
-------	--	--

Related Documents

Chapter 1

Overview

The AMD Media Framework (AMF) delivers comprehensive APIs for video encoding, decoding, and pre- or post-processing that provide access to and benefits from the VCE and UVD hardware blocks and GPU shader-based acceleration.

The AMF is implemented in the form of C++ interfaces. The AMF supports SI and later platforms at full performance and on NI platforms at reduced performance. It supports 32- and 64-bit Windows 7 and Windows 8.1 Desktop applications.

This document describes the AMF APIs that can be used to build a complete multimedia application.

Chapter 2

AMF API

Elementary Data Type

Description Elementary data types are defined to make code potentially portable to other operating systems. A detailed list of Elementary Data types is available in `inc/amf/core/AMFPlatform.h`.

```

typedef      __int64          amf_int64;
typedef      __int32          amf_int32;
typedef      __int16          amf_int16;
typedef      __int8           amf_int8;

typedef      unsigned __int64  amf_uint64;
typedef      unsigned __int32  amf_uint32;
typedef      unsigned __int16  amf_uint16;
typedef      unsigned __int8   amf_uint8;
typedef      size_t           amf_size;

#define AMF_STD_CALL          __stdcall
#define AMF_CDECL_CALL       __cdecl
#define AMF_FAST_CALL        __fastcall
#define AMF_INLINE           inline
#define AMF_FORCEINLINE      __forceinline

typedef      void*            amf_handle;
typedef      double           amf_double;
typedef      float            amf_float;

typedef      void             amf_void;
typedef      bool             amf_bool;
typedef      long             amf_long;
typedef      int              amf_int;
typedef      unsigned long    amf_ulong;
typedef      unsigned int     amf_uint;

typedef      amf_int64        amf_pts; // in 100ns

```

C++ Interfaces

AMF Interface

Description All new objects and components in AMF are implemented in the form of AMF Interfaces. These interfaces implemented in form of abstract C++ classes. Every other interface will be derived from the `AMFInterface` basic interface. It will expose two reference counting methods and a query interface method.

AMF provides default implementation for `AMFInterface` with self-destructing behavior. The SDK also provides a smart pointer template class for easy interface manipulations.

The header file containing the interface definition is located in `inc\amf\core\Interface.h`.

```
class AMFInterface : public AMFObject
```

AMFInterface::Acquire()

Description Increments the reference count for an interface on an object

```
virtual amf_long AMF_STD_CALL Acquire() = 0;
```

AMFInterface::Release()

Description Decrements the reference count for an interface on an object.

```
virtual amf_long AMF_STD_CALL Release() = 0;
```

AMFInterface::QueryInterface()

Description Retrieves pointers to the supported interface on an object.

```
virtual AMF_RESULT AMF_STD_CALL QueryInterface(
    const AMFGuid& interfaceID,
    void** ppInterface) = 0;
```

Parameters

<i>interfaceID [in]</i>	The identifier of the interface being requested.
<i>ppInterface [out]</i>	The address of a pointer variable that receives the interface pointer requested in the <code>interfaceID</code> parameter.

Return Value This method returns `AMF_OK` if the interface is supported, else `AMF_NO_INTERFACE`.

AMFPropertyStorage

Description Most objects in AMF implement `AMFPropertyStorage` or `AMFPropertyStorageEx`. `AMFPropertyStorage` implements property map with string as an ID and `AMFVARIANT` structure as data. There are sets of helper classes and template functions that deal with `AMFVariantStruct` in C++ in a safe way. The default implementation is not thread-safe.

The header file containing the interface definition is located in `inc\amf\core\PropertyStorage.h`.

```
class AMFPropertyStorage : virtual public AMFInterface
```

AMFPropertyStorage::SetProperty()

Description This method is used to set properties on the object.

```
virtual AMF_RESULT AMF_STD_CALL SetProperty(
    const wchar_t* name,
    AMFVariantStruct value) = 0;
```

Parameters

<i>name [in]</i>	The name of the property to be set.
<i>value [in]</i>	The value of the specified property.

Return Value `AMF_OK`.

AMFPropertyStorage::SetProperty()

Description Template method: This method is used to set properties on the object.

```
template<typename T>
AMF_RESULT AMF_STD_CALL SetProperty(
    const wchar_t* name,
    const T& value);
```

Implementation

```
template<typename T> inline
AMF_RESULT AMF_STD_CALL
AMFPropertyStorage::SetProperty(
    const wchar_t* name, const T& value)
{
    AMF_RESULT err = SetProperty(
        name,
        static_cast<const AMFVariantStruct&>(AMFVariant(value)));
    return err;
}
```

Parameters

<i>name [in]</i>	The name of the property to be set.
<i>value [in]</i>	The value of the template argument.

Return Value `AMF_OK`.

AMFPropertyStorage::GetProperty

Description Gets a specific property of the current type.

```
virtual AMF_RESULT AMF_STD_CALL GetProperty(  
const wchar_t* name,  
AMFVariantStruct* pValue) const = 0;
```

Return Value AMF_NOT_FOUND if property not found, else AMF_OK

AMFPropertyStorage::GetProperty

Description Template method: Gets a specific property of the current type.

```
template<typename T>  
AMF_RESULT AMF_STD_CALL GetProperty(  
const wchar_t* name,  
T* pValue) const;
```

Implementation

```
template<typename T> inline  
AMF_RESULT AMF_STD_CALL  
AMFPropertyStorage::GetProperty(  
const wchar_t* name, T* pValue) const  
{  
    AMFVariant var;  
    AMF_RESULT err = GetProperty(name,  
    static_cast<AMFVariantStruct*>(&var));  
    if(err == AMF_OK)  
    {  
        *pValue = static_cast<T>(var);  
    }  
    return err;  
}
```

Parameters

<i>name [in]</i>	The name of the property to get.
<i>pValue [out]</i>	The value of the template argument.

Return Value AMF_NOT_FOUND if property not found, else AMF_OK

AMFPropertyStorage::GetPropertyString

Description Template method: Gets a specific property as a string of the current type.

```
template<typename T>
AMF_RESULT AMF_STD_CALL GetPropertyString(
const wchar_t* name,
T* pValue) const;
```

Implementation

```
template<typename T> inline
AMF_RESULT AMF_STD_CALL
AMFPropertyStorage::GetPropertyString(
const wchar_t* name, T* pValue) const
{
    AMFVariant var;
    AMF_RESULT err = GetProperty(name,
        static_cast<AMFVariantStruct*>(&var));
    if(err == AMF_OK)
    {
        *pValue = var.ToString().c_str();
    }
    return err;
}
```

Parameters

<i>name</i> [in]	The name of the property to get.
<i>pValue</i> [out]	The value of the template argument.

Return Value AMF_NOT_FOUND if property not found, else AMF_OK.

AMFPropertyStorage::GetPropertyWString

Description Template method: Gets a specific property as a wide string of the current type.

```
template<typename T>
AMF_RESULT AMF_STD_CALL GetPropertyWString(
const wchar_t* name,
T* pValue) const;
```

Implementation

```
template<typename T> inline
AMF_RESULT AMF_STD_CALL
AMFPropertyStorage::GetPropertyWString(
const wchar_t* name, T* pValue) const
{
    AMFVariant var;
    AMF_RESULT err = GetProperty(
name,
static_cast<AMFVariantStruct*>(&var));
    if(err == AMF_OK)
    {
        *pValue = var.ToWString().c_str();
    }
    return err;
}
```

Parameters

<i>name</i> [in]	The name of the property to get.
<i>pValue</i> [out]	The value of the template argument.

Return Value AMF_NOT_FOUND if property not found, else AMF_OK.

AMFPropertyStorage::HasProperty()

Description This method checks whether a given property exists.

```
virtual bool AMF_STD_CALL HasProperty(
const wchar_t* name) const = 0;
```

Parameters *name [in]* The name of the property.

Return Value *bool* Returns TRUE if the property exists else returns FALSE.

AMFPropertyStorage::GetPropertyCount()

Description Gets the number of properties.

```
virtual amf_int32 AMF_STD_CALL GetPropertyCount()
const=0;
```

Return Value Returns the number of supported properties.

AMFPropertyStorage::GetPropertyAt()

Description Gets the property at a particular index.

```
virtual AMF_RESULT AMF_STD_CALL GetPropertyAt(
amf_int32 index,
wchar_t* name,
amf_size nameSize,
AMFVariantStruct* pValue) const = 0;
```

Parameters *index [in]* Index to the property to get
name[out] Name of the property at the index
nameSize [out] Size of the property at the index
pValue [out] Pointer to the value of the property

Return Value Returns AMF_OK if property found else returns AMF_INVALID_ARG.

AMFPropertyStorage::Clear()

Description Clears the property values.

```
virtual AMF_RESULT AMF_STD_CALL Clear() = 0;
```

AMFPropertyStorage::AddTo()

Description Adds appropriate properties from “this” object to the pDest object.

```
virtual AMF_RESULT AMF_STD_CALL AddTo(
    AMFPropertyStorage* pDest,
    bool overwrite,
    bool deep) const = 0;
```

Parameters

<i>pDest [in]</i>	Destination Object
<i>overwrite [in]</i>	Controls what to do if the property already exists in pDest
<i>deep [in]</i>	Controls behavior of the properties that are of AMF_VARIANT_INTERFACE type. If true the operation will try to query for AMFCloneable interface and call Clone().

AMFPropertyStorage::CopyTo()

Description Copies appropriate properties from “this” object to the pDest object. Clears the destination properties before copying.

```
virtual AMF_RESULT AMF_STD_CALL CopyTo(
    AMFPropertyStorage* pDest,
    bool deep) const = 0;
```

Parameters

<i>pDest [in]</i>	Destination Object
<i>deep [in]</i>	Controls behavior of the properties that are of AMF_VARIANT_INTERFACE type. If true the operation will try to query for AMFCloneable interface and call Clone().

AMFPropertyStorage::AddObserver()

Description Adds a callback that will be notified when property is changed by calling AMFPropertyStorageObserver::OnPropertyChanged().

```
virtual void AMF_STD_CALL AddObserver(
    AMFPropertyStorageObserver* pObserver) = 0;
```

Parameters

<i>pObserver [in]</i>	Callback.
-----------------------	-----------

AMFPropertyStorage::RemoveObserver()

Description Removes previously added callback by AddObserver().

```
virtual void AMF_STD_CALL RemoveObserver(
    AMFPropertyStorageObserver* pObserver) = 0;
```

Parameters *pObserver [in]* Callback.

AMFPropertyStorageEx

Description Most objects in AMF implement AMFPropertyStorage or AMFPropertyStorageEx. AMFPropertyStorage implements property map with string as an ID and AMFVARIANT structure as data. There are sets of helper classes and template functions that deal with AMFVARIANT in C++ in a safe way. The default implementation is not thread-safe.

The header file containing the interface definition is located in inc\amf\core\PropertyStorageEx.h.

```
class AMFPropertyStorageEx : virtual public AMFPropertyStorage
```

AMFPropertyStorageEx::GetPropertiesInfoCount()

Description Return number of items that are indexed by the first parameter in AMFPropertyStorageEx::GetPropertyInfo().

```
virtual amf_size AMF_STD_CALL
    GetPropertiesInfoCount() const=0;
```

Return Value Number of items that are indexed by first parameter in AMFPropertyStorageEx::GetPropertyInfo().

AMFPropertyStorageEx::GetPropertyInfo()

Description Gets further property information such as the name, description, variant type, property content type, default values, min and max values, access type (Private, Read, Write, Read-Write, Write-Runtime) for a given property.

```
virtual AMF_RESULT AMF_STD_CALL GetPropertyInfo(
    amf_size ind,
    const AMFPropertyInfo** ppInfo)
    const = 0;
```

Parameters

<i>ind [in]</i>	Index to the properties information array
<i>ppInfo [out]</i>	The pointer to the parameter information class.

Return Value AMF_OK.

AMFPropertyStorageEx::GetPropertyInfo()

Description Gets further property information such as the name, description, variant type, property content type, default values, min and max values, access type (Private, Read, Write, Read-Write, Write-Runtime) for a given property.

```
virtual AMF_RESULT AMF_STD_CALL GetPropertyInfo(
    const wchar_t* name,
    const AMFPropertyInfo** ppInfo)
    const=0;
```

Parameters

<i>name [in]</i>	Name of the property for which the information is to be retrieved.
<i>ppInfo [out]</i>	The pointer to the parameter information class.

Return Value AMF_OK if property found else AMF_NOT_FOUND.

AMFPropertyStorageEx::ValidateProperty()

Description This method validates the value of a property.

```
virtual AMF_RESULT AMF_STD_CALL ValidateProperty(
    const wchar_t* name,
    AMFVariantStruct value,
    AMFVariantStruct* pOutValidated) const = 0;
```

Parameters

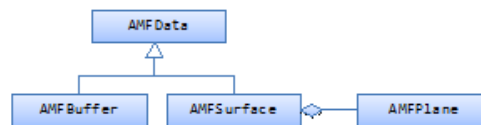
<i>name [in]</i>	The name of the property.
<i>value [in]</i>	The value of the property.
<i>pOutValidated [out]</i>	The validated value of the property.

Return Value AMF_OK if the value is within the range, else AMF_OUT_OF_RANGE in case it's out of range.

2.1 Memory

AMF provides two types of memory object: `AMFBuffer` and `AMFSurface`. Each object implements the common base interface, `AMFData`. Each object is allocated via several factory methods in `AMFContext` (see below). `AMFContext` also provides methods to create `AMFBuffer` or `AMFSurface` based on existing memory objects in various technologies:

- DX9 surfaces (Win Desktop only)
- DX11 Textures
- OpenCL 2D images and buffers (Win Desktop only)
- OpenGL textures (Win Desktop only)
- Host memory



AMFData

Description

Data interface is a base interface for `AMFBuffer` and `AMFSurface` interfaces. It is responsible for storing common properties of video or audio data objects: presentation time, duration, type, etc. Data interface is also responsible for memory type storage and memory conversion.

The header file containing the interface definition located in `inc\amf\core\Data.h`.

```
class AMFData : virtual public AMFPropertyStorage
```

AMFData::GetMemoryType()

Description Returns the memory type of the memory associated with the device.

```
virtual AMF_MEMORY_TYPE AMF_STD_CALL
GetMemoryType() = 0;
```

Return Value

One of the Memory types from the `AMF_MEMORY_TYPE` enumerator.

AMFData::Duplicate()

<i>Description</i>	Duplicates the memory associated with the device to a memory of the type specified by the application.	
	<pre>virtual AMF_RESULT AMF_STD_CALL Duplicate(AMF_MEMORY_TYPE type, AMFData** ppData) = 0;</pre>	
<i>Parameters</i>	<i>type [in]</i>	Type of the duplicate memory.
	<i>ppData [out]</i>	Pointer to the duplicate memory.
<i>Return Value</i>		AMF_OK.

AMFData::Convert()

<i>Description</i>	Convert the memory associated with the device to the 'type' specified by the application. Basically convert the buffer to another memory type using available inter-operation helpers.	
	<pre>virtual AMF_RESULT AMF_STD_CALL Convert(AMF_MEMORY_TYPE type)=0;</pre>	
<i>Parameters</i>	<i>type [in]</i>	Type of the converted memory.
<i>Return Value</i>		Returns AMF_OK if the memory associated with the device is successfully converted to the type specified by the application, else returns the appropriate error value.

AMFData::GetDataType()

<i>Description</i>	Returns the data type of the memory interface	
	<pre>virtual AMF_DATA_TYPE AMF_STD_CALL GetDataType()=0;</pre>	
<i>Return Value</i>		One of the Data types from the AMF_DATA_TYPE enumerator.

AMFData::IsReusable()

<i>Description</i>	Checks whether the memory object is reusable. The object is reusable if allocated by AMF. It is not when the AMF surface or buffer is wrapped around external memory.	
	<pre>virtual bool AMF_STD_CALL IsReusable() = 0;</pre>	
<i>Return Value</i>		TRUE if reusable, else FALSE.

AMFData::SetPts()

Description Sets the presentation timestamp to the memory object.

```
virtual void AMF_STD_CALL SetPts(amf_pts pts) = 0;
```

Parameters *pts [in]* PTS value to be set

AMFData::GetPts()

Description Get the presentation timestamp of the memory object.

```
virtual amf_pts AMF_STD_CALL GetPts() = 0;
```

Return Value PTS value.

AMFData::SetDuration()

Description Sets the duration to the memory object.

```
virtual void AMF_STD_CALL SetDuration(
amf_pts duration) = 0;
```

Parameters *duration [in]* The duration value to be set.

AMFData::GetDuration()

Description Gets the duration for the memory object.

```
virtual void AMF_STD_CALL SetDuration(
amf_pts duration) = 0;
```

Return Value The duration value.

AMFBuffer

Description The `AMFBuffer` interface allows access to unordered memory buffer. This memory object is allocated via several factory methods in `AMFContext`.

The `AMFBuffer` memory object is not thread-safe. It is assumed that a memory object is used by one component at a time.

The header file containing the interface definition located in `inc\amf\core\Buffer.h`.

```
class AMFBuffer : virtual public AMFData
```

AMFBuffer::GetSize()

Description Gets the size of a buffer.

```
virtual amf_sizeAMF_STD_CALL GetSize()=0;
```

Return Value Returns the size of the AMF buffer.

AMFBuffer::GetNative()

Description Allows access to the underlying memory object. Returns pointer to data - for host buffer or `cl_mem` buffer handle - for OpenCL buffer.

All objects that support reference counting do not change the reference count inside this call. The caller should not release the returned object.

```
virtual void* AMF_STD_CALL GetNative()=0;
```

Return Value Pointer to the underlying native memory object

AMFBuffer::AddObserver()

Description Interface to get callbacks from the `AMFPropertyStorageObserver` instance. `OnPropertyChanged` is the callback function executed in case of any property change.

```
virtual void AMF_STD_CALL AddObserver(
    AMFPropertyStorageObserver* pObserver) = 0;
```

Parameters `pObserver [in]` Pointer to the `AMFPropertyStorageObserver` interface.

AMFBuffer::RemoveObserver()

Description Remove the observer.

```
virtual void AMF_STD_CALL RemoveObserver(
    AMFPropertyStorageObserver* pObserver) = 0;
```

Parameters `pObserver [in]` Observer to be removed.

AMFBuffer::AddObserver()

Description Interface to get callbacks from the `AMFBufferObserver` instance. `OnDestroy` is the callback function executed.

```
virtual void AMF_STD_CALL AddObserver(
    AMFBufferObserver* pObserver) = 0;
```

Parameters `pObserver [in]` Pointer to the `AMFBufferObserver`.

AMFBuffer::RemoveObserver()

Description Remove the observer.

```
virtual void AMF_STD_CALL AddObserver(
    AMFBufferObserver* pObserver) = 0;
```

Parameters `pObserver [in]` Observer to be removed.

AMFSurface

Description The `AMFSurface` interface represents video frame and consists of collection of planes. It also holds types of the frame - Progressive, Interlaced.

Surfaces are created via factory methods in the `AMFContext` interface. The factory methods allow one to create allocated surface or attach OpenCL / DX9 / DX9Ex / DX11 / OpenGL objects to an empty surface. The plane objects are created inside surface creation and recreated with every memory type conversion.

The `AMFSurface` memory object is not thread-safe. It is assumed that a memory object is used by one component at a time.

The header file containing the interface definition located in `include\core\Surface.h`.

```
class AMFSurface : virtual public AMFData
```

AMFSurface::GetFormat()

Description Gets the size of the buffer.

```
virtual AMF_SURFACE_FORMAT AMF_STD_CALL
GetFormat() = 0;
```

Return Value Returns the format from the list of supported surface formats as specified in AMF_SURFACE_FORMAT.

AMFSurface::GetPlanesCount()

Description Gets the number of planes.

```
virtual amf_size AMF_STD_CALL GetPlanesCount() = 0;
```

Return Value Number of planes in the surface

AMFSurface::GetPlaneAt()

Description Gets the plane at a particular index.

```
virtual AMFPlane* AMF_STD_CALL GetPlaneAt(
amf_size index)=0;
```

Parameters *index [in]* Index to the plane.

Return Value Pointer to the plane at the specified index.

AMFSurface::GetPlane()

Description Gets the plane of a particular type.

```
virtual AMFPlane* GetPlane(AMF_PLANE_TYPE type)=0;
```

Parameters *type [in]* Type of plane as specified in the AMF_PLANE_TYPE enumerator.

Return Value Pointer to the plane of the specified type.

AMFSurface:: GetFrameType()

Description Gets the type of frame.

```
virtual AMF_FRAME_TYPE AMF_STD_CALL GetFrameType() = 0;
```

Return Value Returns the frame type from the list mentioned in the AMF_FRAME_TYPE enumerator.

AMFSurface::SetFrameType()

<i>Description</i>	Sets the type of the frame.	
	<pre>virtual void AMF_STD_CALL SetFrameType(AMF_FRAME_TYPE type) = 0;</pre>	
<i>Parameters</i>	<i>type [in]</i>	Type of the frame as specified in the AMF_FRAME_TYPE enumerator

AMFSurface::AddObserver()

<i>Description</i>	Interface to get callbacks from the AMFPropertyStorageObserver instance. OnPropertyChanged is the callback function executed in case of any property change.	
	<pre>virtual void AMF_STD_CALL AddObserver(AMFPropertyStorageObserver* pObserver) = 0;</pre>	
<i>Parameters</i>	<i>pObserver [in]</i>	Pointer to the AMFPropertyStorageObserver interface.

AMFSurface::RemoveObserver()

<i>Description</i>	Remove the observer.	
	<pre>virtual void AMF_STD_CALL RemoveObserver(AMFPropertyStorageObserver* pObserver) = 0;</pre>	
<i>Parameters</i>	<i>pObserver [in]</i>	Observer to be removed.

AMFSurface::AddObserver()

<i>Description</i>	Interface to get callbacks from the AMFSurfaceObserver instance. OnSurfaceDataRelease is the callback function executed	
	<pre>virtual void AMF_STD_CALL AddObserver(AMFSurfaceObserver* pObserver) = 0;</pre>	
<i>Parameters</i>	<i>pObserver [in]</i>	Pointer to the AMFBufferObserver.

AMFSurface::RemoveObserver()

<i>Description</i>	Remove the observer.	
	<pre>virtual void AMF_STD_CALL RemoveObserver(AMFSurfaceObserver* pObserver) = 0;</pre>	
<i>Parameters</i>	<i>pObserver [in]</i>	Observer to be removed.

AMFSurface::SetCrop()

<i>Description</i>	Sets the crop parameters.	
	<pre>virtual AMF_RESULT AMF_STD_CALL SetCrop(amf_int32 x, amf_int32 y, amf_int32 width, amf_int32 height) = 0;</pre>	
<i>Parameters</i>	<i>x [in]</i> <i>y [in]</i> <i>width [in]</i> <i>height [in]</i>	Offset in x direction Offset in y direction Crop width Crop height
<i>Return Value</i>	AMF_INVALID_ARG.	

AMFPlane

<i>Description</i>	Plane interface represents two-dimensional array of pixel component values (e.g. Y or UV planes for NV12 surface). It could be part of image for multi-plane image format or whole image for packed image format.
	The <code>AMFPlane</code> memory object is not thread-safe.
	The header file containing the interface definition located in <code>inc\amf\core\Plane.h</code> .
	<pre>class AMFPlane : virtual public AMFInterface</pre>

AMFPlane::GetType()

<i>Description</i>	Gets the type of the plane.
	<pre>virtual AMF_PLANE_TYPE AMF_STD_CALL GetType() = 0;</pre>
<i>Return Value</i>	Type of plane as specified in the <code>AMF_PLANE_TYPE</code> enumerator.

AMFPlane::GetNative()

Description Allows access to the underlying memory object. For plane in surface which is created as DirectX 11 surface, `GetNative()` returns pointer to the `ID3D11Texture2D` interface.

All objects that support reference counting do not change reference count inside this call. The caller should not release the returned object.

```
virtual void* AMF_STD_CALL GetNative()=0;
```

Return Value Pointer to the underlying native memory object.

AMFPlane:: GetPixelSizeInBytes()

Description Gets the pixel size in bytes.

```
virtual amf_int32 AMF_STD_CALL GetPixelSizeInBytes()  
= 0;
```

Return Value Pixel size in bytes.

AMFPlane::GetOffsetX()

Description Gets the offset in the x-coordinate.

```
virtual amf_int32 AMF_STD_CALL GetOffsetX() = 0;
```

Return Value The value of the offset in the x-coordinate.

AMFPlane::GetOffsetY()

Description Gets the offset in the y-coordinate.

```
virtual amf_int32 AMF_STD_CALL GetOffsetY() = 0;
```

Return Value The value of the offset in the y-coordinate.

AMFPlane::GetWidth()

Description Gets the plane width.

```
virtual amf_int32 AMF_STD_CALL GetWidth() = 0;
```

Return Value Width of the plane.

AMFPlane::GetHeight()

Description

Ges the plane height.

```
virtual amf_int32 AMF_STD_CALL    GetHeight() = 0;
```

Return Value

Height of the plane.

AMFPlane:: GetHPitch()

Description

Gets the horizontal pitch for the plane which is a product of the width in pixels and the number of bytes per pixel.

```
virtual amf_int32 AMF_STD_CALL    GetHPitch() = 0;
```

Return Value

Value of the Horizontal pitch.

AMFPlane:: GetVPitch()

Description

Gets the vertical pitch for the plane which is equal to the height in pixels of the plane.

```
virtual amf_int32 AMF_STD_CALL    GetVPitch() = 0;
```

Return Value

Value of the Vertical pitch.

AMFContext

Description

The Context interface is responsible for initializing and storing API-specific data (DX devices, OpenCL contexts). The context provides pairs of Lock/Unlock methods to allow thread-safe access to DX or OpenCL devices. It also provides buffer and surface allocation factory methods.

The context object is thread-safe.

The header file containing the interface definition located in `inc\amf\core\Context.h`.

```
class AMFContext : virtual public AMFPropertyStorage
```

AMFContext:: Terminate()

Description Terminates the context.

```
virtual AMF_RESULT AMF_STD_CALL Terminate() = 0;
```

DirectX 9

AMFContext:: InitDX9()

Description Initializes the DX9 device in the context.

```
virtual AMF_RESULT AMF_STD_CALL InitDX9(
void* pDX9Device) = 0;
```

Parameters *pDX9Device [in]* Valid pointer to DX9 device. If passed NULL, then DX9 is initialized by AMF.

AMFContext:: GetDX9Device ()

Description Get the DirectX9 device.

```
virtual void* AMF_STD_CALL GetDX9Device(
AMF_DX_VERSION dxVersionRequired=AMF_DX9) = 0;
```

Parameters *dxVersionRequired [in]* Version of DirectX - DX9, DX9EX, DX11.0, DX11.1

Return Value Pointer to the DX9 device.

AMFContext:: LockDX9 ()

Description Locks the DX9 device.

```
virtual AMF_RESULT AMF_STD_CALL LockDX9() = 0;
```

Return Value AMF_NOT_INITIALIZED if device is NULL else AMF_OK.

AMFContext:: UnlockDX9 ()

Description Unlocks the DX9 device.

```
virtual AMF_RESULT AMF_STD_CALL UnlockDX9() = 0;
```

Return Value AMF_NOT_INITIALIZED if device is NULL else AMF_OK.

DirectX 11

AMFContext:: InitDX11()

<i>Description</i>	Initializes the DX11 device in the context.	
	<pre>virtual AMF_RESULT AMF_STD_CALL InitDX11(void* pDX11Device, AMF_DX_VERSION dxVersionRequired = AMF_DX11_0) = 0;</pre>	
<i>Parameters</i>	<i>pDX11Device [out]</i>	Valid pointer to DX11 device. If Passed NULL, then DX11 is initialized by AMF.
	<i>dxVersionRequired [in]</i>	DirectX 11 required version.
<i>Return Value</i>	AMF_ALREADY_INITIALIZED if pDX11Device is initialized earlier (pDX11Device != NULL). AMF_OK if device Created and initialized Successfully.	

AMFContext:: GetDX11Device ()

<i>Description</i>	Get the DirectX11 device.	
	<pre>virtual void* AMF_STD_CALL GetDX11Device(AMF_DX_VERSION dxVersionRequired=AMF_DX11_0)=0;</pre>	
<i>Parameters</i>	<i>dxVersionRequire d [in]</i>	Version of DirectX - DX9, DX9EX, DX11.0, DX11.1
<i>Return Value</i>	Pointer to the DX11 device.	

AMFContext:: LockDX11 ()

<i>Description</i>	Locks the DX11 device.	
	<pre>virtual AMF_RESULT AMF_STD_CALL LockDX11() = 0</pre>	
<i>Return Value</i>	AMF_NOT_INITIALIZED if device is NULL else AMF_OK.	

AMFContext:: UnlockDX11 ()

<i>Description</i>	Unlocks the DX11 device.	
	<pre>virtual AMF_RESULT AMF_STD_CALL UnlockDX11() = 0;</pre>	
<i>Return Value</i>	AMF_NOT_INITIALIZED if device is NULL else AMF_OK.	

OpenCL**AMFContext:: InitOpenCL ()**

Description Initializes the OpenCL device in the context.

```
virtual AMF_RESULT AMF_STD_CALL InitOpenCL(
void* pCommandQueue = NULL) = 0;
```

Parameters *pCommandQueue* Pointer to valid command queue.
e [in] If passed NULL, then OpenCL initialized by AMF

Return Value AMF_ALREADY_INITIALIZED if OpenCL device is
initialized earlier (m_pDeviceOpenCL != NULL).
AMF_OK if device Created and initialized successfully.

AMFContext:: GetOpenCLContext ()

Description Gets the OpenCL context.

```
virtual void* AMF_STD_CALL GetOpenCLContext() = 0;
```

Return Value Pointer to the OpenCL context.

AMFContext:: GetOpenCLCommandQueue()

Description Gets the OpenCL command queue.

```
virtual void* AMF_STD_CALL GetOpenCLCommandQueue()=0;
```

Return Value Pointer to the OpenCL command queue.

AMFContext:: GetOpenCLDeviceID ()

Description Gets the OpenCL device ID.

```
virtual void* AMF_STD_CALL GetOpenCLDeviceID() = 0;
```

Return Value Pointer to the OpenCL device ID.

AMFContext:: LockOpenCL ()

Description Locks the OpenCL device.

```
virtual AMF_RESULT AMF_STD_CALL LockOpenCL() = 0;
```

Return Value AMF_NOT_INITIALIZED if device is NULL,
else AMF_OK.

AMFContext:: UnlockOpenCL ()

Description Unlocks the OpenCL device.

```
virtual AMF_RESULT AMF_STD_CALL UnlockOpenCL() = 0;
```

Return Value AMF_NOT_INITIALIZED if device is NULL,
else AMF_OK.

OpenGL

AMFContext:: InitOpenGL ()

Description Initializes the OpenGL device in the context.

```
virtual AMF_RESULT AMF_STD_CALL InitOpenGL(
    amf_handle hOpenGLContext,
    amf_handle hWindow,
    amf_handle hDC) = 0;
```

Parameters *hOpenGLContext [in]* Pointer to OpenGL Context
 hWindow [in] OpenGL Window handle
 hDC [in] Device Context

Return Value AMF_ALREADY_INITIALIZED if OpenGL device is
initialized earlier (m_pDeviceOpenGL!= NULL).

AMF_OK if device Created and initialized
successfully.

AMFContext:: GetOpenGLContext ()

Description Gets the OpenGL context.

```
virtual amf_handle AMF_STD_CALL GetOpenGLContext()=0;
```

Return Value Pointer to the OpenGL context.

AMFContext:: GetOpenGLDrawable ()

Description Gets the OpenGL drawable.

```
virtual amf_handle AMF_STD_CALL GetOpenGLDrawable()=0
```

Return Value Pointer to the OpenGL drawable.

AMFContext:: LockOpenGL ()

Description Locks the OpenGL device.

```
virtual AMF_RESULT AMF_STD_CALL LockOpenGL() = 0;
```

Return Value AMF_NOT_INITIALIZED if device is NULL else
AMF_OK.

AMFContext:: UnlockOpenGL ()

Description Unlocks the OpenGL device.

```
virtual AMF_RESULT AMF_STD_CALL UnlockOpenGL() = 0;
```

Return Value AMF_NOT_INITIALIZED if device is NULL else
AMF_OK.

Buffer Allocation

AMFContext:: AllocBuffer()

Description Creates a buffer and allocates memory for it.

```
virtual AMF_RESULT AMF_STD_CALL AllocBuffer(
    AMF_MEMORY_TYPE type,
    amf_size size,
    AMFBuffer** ppBuffer) = 0;
```

Parameters

<i>type [in]</i>	Memory type of the buffer to be created
<i>size [in]</i>	Size of the buffer
<i>ppBuffer [out]</i>	Pointer to the buffer created

Return Value AMF_OK on success.

AMFContext:: CreateBufferFromHostNative ()

Description Creates AMFBuffer on the passed system memory contiguous block.

```
virtual AMF_RESULT AMF_STD_CALL CreateBufferFromHostNative(
void* pHostBuffer,
amf_size size,
AMFBuffer** ppBuffer,
AMFBufferObserver* pObserver) = 0;
```

Parameters

<i>pHostBuffer [in]</i>	Host memory to which the AMFBuffer is attached.
<i>size [in]</i>	Size of the buffer
<i>ppBuffer [out]</i>	Pointer to the buffer created
<i>pObserver [in]</i>	Observer associated with the memory block

Return Value AMF_OK on success.

Surface Allocation

AMFContext:: AllocSurface()

Description Creates a surface and allocates memory for it.

```
virtual AMF_RESULT AMF_STD_CALL AllocSurface(
AMF_MEMORY_TYPE type,
AMF_SURFACE_FORMAT format,
amf_int32 width,
amf_int32 height,
AMFSurface** ppSurface) = 0;
```

Parameters

<i>type [in]</i>	Memory type of the surface to be created
<i>format [in]</i>	Format of the surface
<i>width [in]</i>	Width in pixels
<i>height [in]</i>	Height in pixels
<i>ppSurface [out]</i>	Pointer to the surface created

Return Value AMF_OK on success.

AMFContext:: CreateSurfaceFromHostNative ()

Description	Creates AMFSurface on passed system memory contiguous block. This version with one width, height, hPitch and vPitch values is applicable only to formats having one plane.	
	<pre>virtual AMF_RESULT AMF_STD_CALL CreateSurfaceFromHostNative(AMF_SURFACE_FORMAT format, amf_int32 width, amf_int32 height, amf_int32 hPitch, amf_int32 vPitch, void* pData, AMFSurface** ppSurface, AMFSurfaceObserver* pObserver) = 0;</pre>	
Parameters	<i>format [in]</i>	Format of the surface
	<i>width [in]</i>	Width in pixels
	<i>height [in]</i>	Height in pixels
	<i>hPitch [in]</i>	Horizontal pitch
	<i>vPitch [in]</i>	Vertical pitch
	<i>pData [in]</i>	Host memory to which the AMFSurface is attached.
	<i>ppSurface [out]</i>	Pointer to the surface created
	<i>pObserver [in]</i>	Observer associated with the memory block
Return Value	AMF_OK on success.	

AMFContext:: CreateSurfaceFromDX9Native ()

Description	Creates AMFSurface on passed DX9 Surface.	
	<pre>AMF_RESULT AMF_STD_CALL CreateSurfaceFromDX9Native(void* pDX9Surface, AMFSurface** ppSurface, AMFSurfaceObserver* pObserver) = 0;</pre>	
Parameters	<i>pDX9Surface [in]</i>	DX9 Surface to which the AMFSurface is attached.
	<i>ppSurface [out]</i>	Pointer to the surface created
	<i>pObserver [in]</i>	Observer associated with the memory block. This is to get a notification when attached resources could be released.
Return Value	AMF_OK on success.	

AMFContext:: CreateSurfaceFromDX11Native ()

Description Creates AMFSurface on passed DX11 Texture.

```
virtual AMF_RESULT AMF_STD_CALL CreateSurfaceFromDX11Native(
void* pDX11Surface,
AMFSurface** ppSurface,
AMFSurfaceObserver* pObserver) = 0
```

Parameters

<i>pDX11Surface</i> [in]	DX11 texture to which the AMFSurface is attached.
<i>ppSurface</i> [out]	Pointer to the surface created
<i>pObserver</i> [in]	Observer associated with the memory block. This is to get a notification when attached resources could be released.

Return Value AMF_OK on success.

AMFContext:: CreateSurfaceFromOpenCLNative()

Description Creates AMFSurface on passed OpenCL object.

```
virtual AMF_RESULT AMF_STD_CALL
CreateSurfaceFromOpenCLNative (
AMF_SURFACE_FORMAT format,
amf_int32 width,
amf_int32 height,
void** pCLPlanes,
AMFSurface** ppSurface,
AMFSurfaceObserver* pObserver ) = 0;
```

Parameters

<i>format</i> [in]	Surface format
<i>width</i> [in]	width in pixels
<i>height</i> [in]	Height in pixels
<i>pCLPlanes</i> []	OpenCL handle to which the AMFSurface is attached.
<i>ppSurface</i> [out]	Pointer to the surface created
<i>pObserver</i> [in]	Observer associated with the memory block. This is to get notification when attached resources could be released.

Return Value AMF_OK on success.

AMFComponent

Description Each component implements the common `AMFComponent` interface. The `AMFComponent` interface is derived from `AMFPropertyStorageEx`. All components are thread-safe.

The header file containing the interface definition located in `inc\amf\core\Component.h`.

```
class AMFComponent : virtual public AMFPropertyStorageEx
```

AMFComponent:: Init()

Description Initializes the AMF component.

```
virtual AMF_RESULT AMF_STD_CALL Init(
    AMF_SURFACE_FORMAT format,
    amf_int32 width,
    amf_int32 height) = 0;
```

Parameters	<i>format [in]</i>	Format
	<i>width [in]</i>	Width in pixels
	<i>height [in]</i>	Height in pixels

Return Value	<code>AMF_OK</code> on success.
---------------------	---------------------------------

AMFComponent:: ReInit ()

Description Reinitializes the `AMFComponent` with updated input parameters

```
virtual AMF_RESULT AMF_STD_CALL ReInit(
    amf_int32 width,
    amf_int32 height) = 0;
```

Parameters	<i>width [in]</i>	Width in pixels
	<i>height [in]</i>	Height in pixels

Return Value	If successful returns <code>AMF_OK</code> .
---------------------	---

AMFComponent:: Terminate ()

Description Destroys the AMF component. Releases all the internal resources.

```
virtual AMF_RESULT AMF_STD_CALL Terminate() = 0;
```

Return Value If successful returns AMF_OK.

AMFComponent:: Drain ()

Description Drains the AMF component. Returns all the accumulated resources.

```
virtual AMF_RESULT AMF_STD_CALL Drain() = 0;
```

Return Value If successful returns AMF_OK.

AMFComponent:: Flush ()

Description Flush the AMF component.

```
virtual AMF_RESULT AMF_STD_CALL Flush() = 0;
```

Return Value If successful returns AMF_OK.

AMFComponent:: SubmitInput ()

Description Submits input data to the AMF component.

```
virtual AMF_RESULT AMF_STD_CALL SubmitInput (
    AMFData* pData) = 0;
```

Parameters *pData [in]* AMFData fed to the component to be processed

Return Value If successful returns AMF_OK.

AMFComponent:: QueryOutput ()

Description Query the component for the output result.

```
virtual AMF_RESULT AMF_STD_CALL QueryOutput (
    AMFData** ppData) = 0;
```

Parameters *ppData [out]* Pointer to the buffer outputted by the component

Return Value If successful returns AMF_OK.

AMFComponent:: GetContext ()

<i>Description</i>	Gets the context that the component is associated with.
	<code>virtual AMFContext* AMF_STD_CALL GetContext() = 0;</code>
<i>Return Value</i>	The context that the component is associated with.

AMFComponent:: SetOutputDataAllocatorCB ()

<i>Description</i>	Sets the callback function for the output data.
	<code>virtual AMF_RESULT AMF_STD_CALL SetOutputDataAllocatorCB(AMFDataAllocatorCB* callback) = 0;</code>
<i>Parameters</i>	<i>callback [in]</i> The callback function to be set.

AMFDataAllocatorCB

<i>Description</i>	Class to implement Buffer and Surface allocation for a component.
	The header file containing the interface definition located in <code>inc\amf\core\Component.h</code> .
	<code>class AMFDataAllocatorCB : virtual public AMFInterface</code>

AMFDataAllocatorCB : AllocBuffer

<i>Description</i>	Allocates the AMF buffer.
	<code>virtual AMF_RESULT AMF_STD_CALL AllocBuffer(AMF_MEMORY_TYPE type, amf_size size, AMFBuffer** ppBuffer) = 0;</code>
<i>Parameters</i>	<i>type [in]</i> Memory Type <i>size [in]</i> Size of memory to be allocated <i>ppBuffer [out]</i> Allocated AMF Buffer pointer
<i>Return Value</i>	AMF_OK on success.

AMFDataAllocatorCB : AllocSurface

<i>Description</i>	Allocates AMF Surface.														
	<pre>virtual AMF_RESULT AMF_STD_CALL AllocSurface(AMF_MEMORY_TYPE type, AMF_SURFACE_FORMAT format, amf_int32 width, amf_int32 height, amf_int32 hPitch, amf_int32 vPitch, AMFSurface** ppSurface) = 0;</pre>														
<i>Parameters</i>	<table border="0"> <tr> <td><i>type [in]</i></td> <td>Memory Type</td> </tr> <tr> <td><i>format [in]</i></td> <td>AMF Surface format</td> </tr> <tr> <td><i>width [in]</i></td> <td>Width of surface in pixels</td> </tr> <tr> <td><i>height [in]</i></td> <td>Height of surface in pixels</td> </tr> <tr> <td><i>hPitch [in]</i></td> <td>Pitch in horizontal direction</td> </tr> <tr> <td><i>Pitch [in]</i></td> <td>vPitch in Vertical direction</td> </tr> <tr> <td><i>ppSurface [out]</i></td> <td>Allocated AMF Surface pointer</td> </tr> </table>	<i>type [in]</i>	Memory Type	<i>format [in]</i>	AMF Surface format	<i>width [in]</i>	Width of surface in pixels	<i>height [in]</i>	Height of surface in pixels	<i>hPitch [in]</i>	Pitch in horizontal direction	<i>Pitch [in]</i>	vPitch in Vertical direction	<i>ppSurface [out]</i>	Allocated AMF Surface pointer
<i>type [in]</i>	Memory Type														
<i>format [in]</i>	AMF Surface format														
<i>width [in]</i>	Width of surface in pixels														
<i>height [in]</i>	Height of surface in pixels														
<i>hPitch [in]</i>	Pitch in horizontal direction														
<i>Pitch [in]</i>	vPitch in Vertical direction														
<i>ppSurface [out]</i>	Allocated AMF Surface pointer														
<i>Return Value</i>	If successful returns AMF_OK.														

Component Capability

AMFCaps

<i>Description</i>	<p>This is the base interface for every hardware module supported. Every module-specific interface must extend this interface.</p> <p>The header file containing the interface definition located in <code>inc\amf\component\ComponentCaps.h</code>.</p> <pre>class AMFCaps : public virtual AMFInterface</pre>
--------------------	---

AMFCaps: GetAccelerationType

<i>Description</i>	<p>Get the acceleration type for a component.</p> <pre>virtual AMF_ACCELERATION_TYPE AMF_STD_CALL GetAccelerationType() const = 0;</pre>
<i>Return Value</i>	Returns one of the acceleration types defined in <code>AMF_ACCELERATION_TYPE</code> enumeration (Not supported, Hardware, GPU, or Software).

AMFCaps: GetInputCaps

<i>Description</i>	Get input capabilities of a component.	
	<pre>virtual AMF_RESULT AMF_STD_CALL GetInputCaps (AMFIOCaps** input) = 0;</pre>	
<i>Parameters</i>	<i>input [out]</i>	Pointer to AMFIOCaps containing component Capability information.
<i>Return Value</i>	AMF_INVALID_ARG if input is NULL AMF_OK if input capabilities of the component are successfully retrieved.	

AMFCaps: GetOutputCaps

<i>Description</i>	Get output capabilities of a component.	
	<pre>virtual AMF_RESULT AMF_STD_CALL GetOutputCaps (AMFIOCaps** output) = 0;</pre>	
<i>Parameters</i>	<i>output [out]</i>	Pointer to AMFIOCaps containing component Capability information.
<i>Return Value</i>	AMF_INVALID_ARG if input is NULL AMF_OK if output capabilities of the component are successfully retrieved.	

AMFIOCaps

Description Interface containing input as well as output capabilities for a component.

The header file containing the interface definition located in
inc\amf\component\ComponentCaps.h.

```
class AMFIOCaps : public virtual AMFInterface
```

AMFIOCaps: GetWidthRange

Description Get the width range supported by the component in pixels.

```
virtual void AMF_STD_CALL GetWidthRange(
    amf_int32* minWidth,
    amf_int32* maxWidth) const = 0;
```

Parameters

<i>minWidth [out]</i>	Min supported width in pixels
<i>maxWidth [out]</i>	Max supported width in pixels

AMFIOCaps: GetHeightRange

Description Get the height range supported by the component in pixels.

```
virtual void AMF_STD_CALL GetHeightRange(
    amf_int32* minHeight,
    amf_int32* maxHeight) const = 0;
```

Parameters

<i>iminHeight [out]</i>	Min supported height in pixels
<i>maxHeight [out]</i>	Max supported height in pixels

AMFIOCaps: GetVertAlign

Description Get memory alignment in lines. Vertical alignment must be multiples of this number.

```
virtual amf_int32 AMF_STD_CALL GetVertAlign() const = 0;
```

Return Value Vertical alignment in number of lines.

AMFIOCaps: GetNumOfFormats

Description Get the number of surface formats supported.

```
virtual amf_int32 AMF_STD_CALL GetNumOfFormats() const = 0;
```

Return Value Returns the number of supported surface formats from the list of formats defined in AMF_SURFACE_FORMAT.

AMFIOCaps: GetFormatAt

Description

Get the surface format for a particular index.

```
virtual AMF_RESULT AMF_STD_CALL GetFormatAt(
    amf_int32 index,
    AMF_SURFACE_FORMAT* format,
    amf_bool* native) const = 0;
```

Parameters

<i>index [in]</i>	index to list of surface formats
<i>format [out]</i>	format at the index specified
<i>native [out]</i>	bool value if the format is native or not

Return Value

AMF_OK if successful else returns
 AMF_INVALID_ARG if
 Index is out of range.

AMFIOCaps: GetNumOfMemoryTypes

Description

Gets the number of memory types supported.

```
virtual amf_int32 AMF_STD_CALL GetNumOfMemoryTypes()
const = 0;
```

Return Value

Returns the number of supported memory
 types from the list of types defined in
 AMF_MEMORY_TYPE.

AMFIOCaps: GetMemoryTypeAt

Description Get the memory type for a particular index.

```
virtual AMF_RESULT AMF_STD_CALL GetMemoryTypeAt (
    amf_int32 index,
    AMF_MEMORY_TYPE* memType,
    amf_bool* native) const = 0;
```

Parameters

<i>index [in]</i>	index to list of memory type
<i>memType [out]</i>	memory type at the index specified
<i>native [out]</i>	bool value if the memory type is native or not

Return Value AMF_OK if successful else returns AMF_INVALID_ARG if Index is out of range.

AMFIOCaps: IsInterlacedSupported

Description Gets information about whether interlaced is supported or not.

```
virtual amf_bool AMF_STD_CALL IsInterlacedSupported()
const = 0;
```

Return Value Returns TRUE if supported else returns FALSE.

AMFDecoderCaps

Description Class to implement methods to understand decoder capability.

The header file containing the interface definition located in inc\amf\component\VideoDecoderCaps.h.

```
class AMFDecoderCaps : public virtual AMFCaps
```

AMFDecoderCaps: GetMaxNumOfStreams

Description Method which returns the max. no of streams that can be decoded in parallel by the UVD.

```
virtual amf_int32 AMF_STD_CALL GetMaxNumOfStreams() const = 0;
```

Return Value Max. no of streams that can be decoded in parallel by the Hardware.

AMFEncoderCaps

Description Class to implement methods to understand encoder capability.

The header file containing the interface definition located in inc\amf\component\VideoEncoderCaps.h.

```
class AMFEncoderCaps : public virtual AMFCaps
```

AMFEncoderCaps: GetMaxBitrate

Description Method which returns the max. bitrate supported by the encoder.

```
virtual amf_uint32 GetMaxBitrate() const = 0;
```

Return Value Max. bitrate supported by the encoder.

AMFEncoderCaps: GetMaxNumOfStreams

Description Method which returns the max. no of streams that can be encoded in parallel by the VCE.

```
virtual amf_int32 AMF_STD_CALL GetMaxNumOfStreams() const = 0;
```

Return Value Max. no of streams that can be encoded in parallel by the hardware.

AMFH264EncoderCaps

Description Class to implement methods to understand the H.264 encoder capability.

The header file containing the interface definition located in
inc\amf\component\VideoEncoderVCECaps.h.

```
class AMFH264EncoderCaps : public AMFEncoderCaps
```

AMFH264EncoderCaps: GetNumOfSupportedProfiles

Description Get the number of profiles supported by the H.264 Encoder.

```
virtual amf_int32 GetNumOfSupportedProfiles() const = 0;
```

Return Value Number of supported profiles.

AMFH264EncoderCaps: GetProfile

Description Get the profile for the specified index.

```
virtual AMF_VIDEO_ENCODER_PROFILE_ENUM  
GetProfile(amf_int32 index) const = 0;
```

Parameters *index [in]* Index to profiles mentioned in
AMF_VIDEO_ENCODER_PROFILE_ENUM

Return Value Profile enum value.

AMFH264EncoderCaps: GetNumOfSupportedLevels

Description Get the number of levels supported by the H.264 Encoder.

```
virtual amf_int32 GetNumOfSupportedLevels() const = 0;
```

Return Value Number of supported levels.

AMFH264EncoderCaps: GetLevel

Description Get the level for the specified index.

```
virtual amf_uint32 GetLevel(amf_int32 index) const = 0;
```

Parameters *index [in]* Index.

Return Value Level value.

AMFH264EncoderCaps: GetNumOfRateControlMethods

Description Get the number of rate control methods supported by the H.264 Encoder.

```
virtual amf_int32 GetNumOfRateControlMethods() const = 0;
```

Return Value Number of supported rate control methods.

AMFH264EncoderCaps: GetRateControlMethod

Description Get the rate control method for the specified index.

```
virtual AMF_VIDEO_ENCODER_RATE_CONTROL_METHOD_ENUM  
GetRateControlMethod(amf_int32 index) const = 0;
```

Parameters *index [in]* Index to rate control methods mentioned in the AMF_VIDEO_ENCODER_RATE_CONTROL_METHOD_ENUM enumerator.

Return Value Rate control method enum value.

AMFH264EncoderCaps: GetMaxSupportedJobPriority

Description Get the max supported job priority.

```
virtual H264EncoderJobPriority
GetMaxSupportedJobPriority() const = 0;
```

Return Value Max. supported job priority from the list of priorities specified in the H264EncoderJobPriority enum.

AMFH264EncoderCaps: IsBPictureSupported

Description Method to check whether BPictures is supported or not.

```
virtual amf_bool IsBPictureSupported() const = 0;
```

Return Value Returns TRUE if supported else returns FALSE.

AMFH264EncoderCaps: GetNumOfReferenceFrames

Description Gets the range for the number of supported reference frames.

```
virtual void GetNumOfReferenceFrames(
amf_uint32* minNum,
amf_uint32* maxNum) const = 0;
```

Parameters

<i>minNum [out]</i>	min number of supported reference frames
<i>maxNum [out]</i>	max number of supported reference frames

Return Value

AMFH264EncoderCaps: CanOutput3D

Description Check whether the encoder can output 3D content.

```
virtual amf_bool CanOutput3D() const = 0;
```

Return Value Returns TRUE if 3D output is supported else returns FALSE.

AMFH264EncoderCaps: GetMaxNumOfTemporalLayers

Description Get the maximum number of temporal enhancement layers supported by the encoder.

```
virtual amf_int32 GetMaxNumOfTemporalLayers() const = 0;
```

Return Value Maximum number of temporal enhancement layers.

Debug

AMFTraceWriter

Description Class to implement an interface for custom trace writer. The header file containing the interface definition is located in inc\amf\core\Debug.h.

```
class AMFTraceWriter;
```

AMFTraceWriter: Write

Description Writes a string to a stream.

```
virtual void Write(const wchar_t* scope, const wchar_t* message) = 0;
```

Parameter *message [in]* Output string.

AMFTraceWriter: Flush

Description Flush the stream.

```
virtual void Flush() = 0;
```

AMFTraceSetPath

Description Sets trace path.

```
AMF_CORE_LINK AMF_RESULT AMF_CDECL_CALL AMFTraceSetPath(const
wchar_t* path);
```

Parameters *path [in]* Trace path to be set.

Return Value AMF_OK if successful, else AMF_FAIL.

AMFTraceGetPath

Description Gets trace path.

```
AMF_CORE_LINK AMF_RESULT AMF_CDECL_CALL AMFTraceGetPath (
    wchar_t* path,
    amf_size* pSize
);
```

Parameters *path [out]* buffer able to hold *pSize symbols; path is copied there.

pSize [in, out] Size of buffer, returned needed size of buffer including zero terminator.

Return Value AMF_OK if successful.

AMFTraceEnableWriter

<i>Description</i>	Enable/Disable trace to registered writer.	
	<pre>AMF_CORE_LINK bool AMF_CDECL_CALL AMFTraceEnableWriter(const wchar_t* writerID, bool enable);</pre>	
<i>Parameters</i>	<i>writerID [in]</i> <i>enable [in]</i>	writer ID If true and writer not enabled, insert writer to the list of enabled writers. If false and writer already enabled, Erase writer from list of enabled writers.
<i>Return Value</i>	Previous state: True if writer already enabled, else false.	

AMFTraceWriterEnabled

<i>Description</i>	Return flag if writer enabled.	
	<pre>AMF_CORE_LINK bool AMF_CDECL_CALL AMFTraceWriterEnabled(const wchar_t* writerID);</pre>	
<i>Parameters</i>	<i>writerID [in]</i>	Writer ID
<i>Return Value</i>	True if already enabled, else false.	

AMFTraceSetGlobalLevel

<i>Description</i>	Sets trace level for writer and scope.	
	<pre>AMF_CORE_LINK amf_int32 AMF_CDECL_CALL AMFTraceSetGlobalLevel(amf_int32 level);</pre>	
<i>Parameters</i>	<i>level [in]</i>	Trace level.
<i>Return Value</i>	Previous level value.	

AMFTraceGetGlobalLevel

<i>Description</i>	Gets the trace level.	
	<pre>AMF_CORE_LINK amf_int32 AMF_CDECL_CALL AMFTraceGetGlobalLevel();</pre>	
<i>Return Value</i>	Returns global trace level.	

AMFTraceSetWriterLevel

<i>Description</i>	Sets trace level for writer.	
	<pre>AMF_CORE_LINK amf_int32 AMF_CDECL_CALL AMFTraceSetWriterLevel(const wchar_t* writerID, amf_int32 level);</pre>	
<i>Parameters</i>	<i>writerID [in]</i> <i>level [in]</i>	writer ID Trace level
<i>Return Value</i>	Returns previous settings.	

AMFTraceGetWriterLevel

<i>Description</i>	Gets trace level for writer.	
	<pre>AMF_CORE_LINK amf_int32 AMF_CDECL_CALL AMFTraceGetWriterLevel(const wchar_t* writerID);</pre>	
<i>Parameters</i>	<i>writerID [in]</i>	Writer ID.
<i>Return Value</i>	Trace level for the requested writer.	

AMFTraceSetWriterLevelForScope

<i>Description</i>	Sets trace level for writer and scope.	
	<pre>AMF_CORE_LINK amf_int32 AMF_CDECL_CALL AMFTraceSetWriterLevelForScope(const wchar_t* writerID, const wchar_t* scope, amf_int32 level);</pre>	
<i>Parameters</i>	<i>writer [in]</i>	Writer ID
	<i>scope [in]</i>	Scope value
	<i>level [in]</i>	Trace level to be set for writer and associated scope.
<i>Return Value</i>	Returns previous settings.	

AMFTraceGetWriterLevelForScope

<i>Description</i>	Gets trace level for writer and scope.	
	<pre>AMF_CORE_LINK amf_int32 AMF_CDECL_CALL AMFTraceGetWriterLevelForScope(const wchar_t* writerID, const wchar_t* scope);</pre>	
<i>Parameters</i>	<i>writerID [in]</i>	ID of the writer.
	<i>scope [in]</i>	Scope value.
<i>Return Value</i>	Trace level value.	

AMFTraceRegisterWriter

<i>Description</i>	Register custom trace writer.	
	<pre>AMF_CORE_LINK void AMF_CDECL_CALL AMFTraceRegisterWriter(const wchar_t* writerID, AMFTraceWriter* pWriter);</pre>	
<i>Parameters</i>	<i>writerID [in]</i>	Trace writer ID.
	<i>pWritec</i>	Custom trace writer.

AMFTraceUnregisterWriter

<i>Description</i>	Enable performance monitoring logging.
	<code>AMF_CORE_LINK void AMF_CDECL_CALL AMFEnablePerformanceMonitor(bool enable);</code>
<i>Parameters</i>	<i>enable [in]</i> True to enable performance monitoring logging.

AMFEnablePerformanceMonitor

<i>Description</i>	Sets trace path.
	<code>AMF_CORE_LINK AMF_RESULT AMF_CDECL_CALL AMFTraceSetPath(const wchar_t* path);</code>
<i>Parameters</i>	<i>path [in]</i> Trace path to be set.

AMFPerformanceMonitorEnabled

<i>Description</i>	Check whether performance monitoring is enabled or disabled.
	<code>AMF_CORE_LINK bool AMF_CDECL_CALL AMFPerformanceMonitorEnabled();</code>
<i>Return Value</i>	True if performance monitoring is enabled, else false.

AMFAssertsEnable

<i>Description</i>	Enable asserts in checks.
	<code>AMF_CORE_LINK void AMF_CDECL_CALL AMFAssertsEnable(bool enable);</code>
<i>Parameters</i>	<i>enable [in]</i> Enables or disables asserts in checks.

AMFAssertsEnabled

<i>Description</i>	Returns true if asserts in checks are enabled.
	<code>AMF_CORE_LINK bool AMF_CDECL_CALL AMFAssertsEnabled();</code>
<i>Return Value</i>	Returns true if asserts in checks are enabled.

Chapter 3

Enumerations

Memory-related enumerations

The header file containing the memory enumeration definition is located in `inc\amf\core\Data.h`, `inc\amf\core\Surface.h`, `inc\amf\core\Plain.h`.

AMF_MEMORY_TYPE

Description

```
enum AMF_MEMORY_TYPE
{
    AMF_MEMORY_UNKNOWN = 0,
    AMF_MEMORY_HOST = 1,
    AMF_MEMORY_DX9 = 2,
    AMF_MEMORY_DX11 = 3,
    AMF_MEMORY_OPENCL = 4,
    AMF_MEMORY_OPENGL = 5,

};
```

AMF_DATA_TYPE

Description

```
enum AMF_DATA_TYPE
{
    AMF_DATA_BUFFER = 0,
    AMF_DATA_SURFACE = 1,
    AMF_DATA_AUDIO_BUFFER = 2,
    AMF_DATA_USER = 1000,
    // all extensions will be AMF_DATA_USER + i
};
```

Memory-related enumerations

The header file containing the memory enumeration definition is located in
inc\amf\core\Data.h, inc\amf\core\Surface.h, inc\amf\core\Plain.h.

AMF_SURFACE_FORMAT

Description

```
enum
AMF_SURFACE_FORMAT
{
AMF_SURFACE_UNKNOWN = // < 1 - planar Y width x height + packed UV
0, width/2 x // height/2. 8 bits per component
AMF_SURFACE_NV12, //< 2 - planar Y width x height + V width/2 x
height/2 // + U width/2 x height/2. 8 bits per
component
AMF_SURFACE_YV12, //< 3 - packed - 8 bit per component
//< 4 - packed - 8 bit per component
AMF_SURFACE_BGRA, //< 5 - packed - 8 bit per component
AMF_SURFACE_ARGB, //< 6 - single component - 8 bit
AMF_SURFACE_RGBA, //< 7 - planar Y width x height +
AMF_SURFACE_GRAY8, // U width/2 x height/2 + V width/2 x height/2.
AMF_SURFACE_YUV420P, // 8 bit per component

//< 8 - double component - 8 bit per component
AMF_SURFACE_U8V8, //< 9 - YUY2: Byte 0=8-bit Y'0;
AMF_SURFACE_YUY2, // Byte 1=8-bit Cb;
// Byte 2=8-bit Y'1;
// Byte 3=8-bit Cr

AMF_SURFACE_FIRST =
AMF_SURFACE_NV12,

AMF_SURFACE_LAST =
AMF_SURFACE_YUY2
};
```

AMF_PLANE_TYPE

Description

```
enum AMF_PLANE_TYPE
{
AMF_PLANE_UNKNOWN = 0,
AMF_PLANE_PACKED = 1, // for all packed formats: BGRA, YUY2
AMF_PLANE_Y = 2,
AMF_PLANE_UV = 3,
AMF_PLANE_U = 4,
AMF_PLANE_V = 5,
};
```

**Memory-related
enumerations**

The header file containing the memory enumeration definition is located in
`inc\amf\core\Data.h, inc\amf\core\Surface.h, inc\amf\core\Plain.h.`

AMF_FRAME_TYPE

Memory-related enumerations

The header file containing the memory enumeration definition is located in
inc\amf\core\Data.h, inc\amf\core\Surface.h, inc\amf\core\Plain.h.

Description

Memory-related enumerations

The header file containing the memory enumeration definition is located in
inc\amf\core\Data.h, inc\amf\core\Surface.h, inc\amf\core\Plain.h.

```
enum AMF_FRAME_TYPE
{
    // flags
    AMF_FRAME_STEREO_FLAG = 0x10000000,
    AMF_FRAME_LEFT_FLAG = AMF_FRAME_STEREO_FLAG | 0x20000000,
    AMF_FRAME_RIGHT_FLAG = AMF_FRAME_STEREO_FLAG | 0x40000000,
    AMF_FRAME_BOTH_FLAG = AMF_FRAME_LEFT_FLAG | AMF_FRAME_RIGHT_FLAG,
    AMF_FRAME_INTERLEAVED_FLAG = 0x01000000,
    AMF_FRAME_FIELD_FLAG = 0x02000000,
    AMF_FRAME_EVEN_FLAG = 0x04000000,
    AMF_FRAME_ODD_FLAG = 0x08000000,

    // values
    AMF_FRAME_UNKNOWN = -1,
    AMF_FRAME_PROGRESSIVE = 0,
    AMF_FRAME_INTERLEAVED_EVEN_FIRST =
    AMF_FRAME_INTERLEAVED_FLAG |
    AMF_FRAME_EVEN_FLAG,

    AMF_FRAME_INTERLEAVED_ODD_FIRST =
    AMF_FRAME_INTERLEAVED_FLAG | AMF_FRAME_ODD_FLAG,

    AMF_FRAME_FIELD_SINGLE_EVEN =
    AMF_FRAME_FIELD_FLAG | AMF_FRAME_EVEN_FLAG,
    AMF_FRAME_FIELD_SINGLE_ODD =
    AMF_FRAME_FIELD_FLAG | AMF_FRAME_ODD_FLAG,

    AMF_FRAME_STEREO_LEFT = AMF_FRAME_LEFT_FLAG,
    AMF_FRAME_STEREO_RIGHT = AMF_FRAME_RIGHT_FLAG,
    AMF_FRAME_STEREO_BOTH = AMF_FRAME_BOTH_FLAG,

    AMF_FRAME_INTERLEAVED_EVEN_FIRST_STEREO_LEFT =
    AMF_FRAME_INTERLEAVED_FLAG | AMF_FRAME_EVEN_FLAG |
    AMF_FRAME_LEFT_FLAG,

    AMF_FRAME_INTERLEAVED_EVEN_FIRST_STEREO_RIGHT =
    AMF_FRAME_INTERLEAVED_FLAG | AMF_FRAME_EVEN_FLAG |
    AMF_FRAME_RIGHT_FLAG,

    AMF_FRAME_INTERLEAVED_EVEN_FIRST_STEREO_BOTH =
    AMF_FRAME_INTERLEAVED_FLAG | AMF_FRAME_EVEN_FLAG |
    AMF_FRAME_BOTH_FLAG,

    AMF_FRAME_INTERLEAVED_ODD_FIRST_STEREO_LEFT =
    AMF_FRAME_INTERLEAVED_FLAG | AMF_FRAME_ODD_FLAG |
    AMF_FRAME_LEFT_FLAG,

    AMF_FRAME_INTERLEAVED_ODD_FIRST_STEREO_RIGHT =
    AMF_FRAME_INTERLEAVED_FLAG | AMF_FRAME_ODD_FLAG |
    AMF_FRAME_RIGHT_FLAG,

    AMF_FRAME_INTERLEAVED_ODD_FIRST_STEREO_BOTH =
    AMF_FRAME_INTERLEAVED_FLAG | AMF_FRAME_ODD_FLAG |
    AMF_FRAME_BOTH_FLAG,

};
```

Enumerations

Copyright © 2014 Advanced Micro Devices, Inc. All rights reserved.

Memory-related enumerations

The header file containing the memory enumeration definition is located in
inc\amf\core\Data.h, inc\amf\core\Surface.h, inc\amf\core\Plain.h.

Result-related enumerations

The header file containing the AMF result enumerations definition is located in
inc\amf\core\Result.h.

AMF_RESULT

Description

```
enum AMF_RESULT
{
    AMF_OK = 0,
    AMF_FAIL,

    // common errors
    AMF_UNEXPECTED,
    AMF_ACCESS_DENIED,
    AMF_INVALID_ARG,
    AMF_OUT_OF_RANGE,
    AMF_OUT_OF_MEMORY,
    AMF_INVALID_POINTER,
    AMF_NO_INTERFACE,
    AMF_NOT_IMPLEMENTED,
    AMF_NOT_SUPPORTED,
    AMF_NOT_FOUND,
    AMF_ALREADY_INITIALIZED,
    AMF_NOT_INITIALIZED,
    AMF_INVALID_FORMAT, // invalid data format
    AMF_WRONG_STATE,
    AMF_FILE_NOT_OPEN, // cannot open file

    // device common codes
    AMF_NO_DEVICE,

    // device directx
    AMF_DIRECTX_FAILED,

    // device opengl
    AMF_OPENGL_FAILED,

    // device opengl
    AMF_GLX_FAILED, //failed to use GLX

    // device alsa
    AMF_ALSA_FAILED, //failed to use ALSA
}
```


Result-related enumerations

The header file containing the AMF result enumerations definition is located in `inc\amf\core\Result.h`.

```
// component common codes

//result codes
AMF_EOF,
AMF_REPEAT, //returned by AMFComponent::QueryOutput if
AMF_NEED_MORE_INPUT, //more frames to be submitted

AMF_INPUT_FULL, //returned by AMFComponent::SubmitInput if
//input queue is full

AMF_RESOLUTION_CHANGED, //resolution changed client needs to
//Drain/Terminate/Init

//error codes
AMF_INVALID_DATA_TYPE, //invalid data type
AMF_INVALID_RESOLUTION, //invalid resolution (width or height)
AMF_CODEC_NOT_SUPPORTED, //codec not supported
AMF_SURFACE_FORMAT_NOT_SUPPORTED, //surface format not supported

//component video decoder
AMF_DECODER_NOT_PRESENT, //failed to create the decoder
AMF_DECODER_SURFACE_ALLOCATION_FAILED, //failed to create the surface for decoding
AMF_DECODER_NO_FREE_SURFACES,

//component video encoder //failed to create the encoder
AMF_ENCODER_NOT_PRESENT,

//component dem
AMF_DEM_ERROR,
AMF_DEM_PROPERTY_READONLY,
AMF_DEM_REMOTE_DISPLAY_CREATE_FAILED,
AMF_DEM_START_ENCODING_FAILED,
AMF_DEM_QUERY_OUTPUT_FAILED,
};
```

Video Encoder enumerations

The header file containing the Video Encoder enumerations definition is located in `inc\amf\components\VideoEncoderVCE.h`.

AMF_VIDEO_ENCODER_USAGE_ENUM

Description

```
enum AMF_VIDEO_ENCODER_USAGE_ENUM
{
    AMF_VIDEO_ENCODER_USAGE_TRANSCODING = 0,
    AMF_VIDEO_ENCODER_USAGE_ULTRA_LOW_LATENCY,
    AMF_VIDEO_ENCODER_USAGE_LOW_LATENCY,
    AMF_VIDEO_ENCODER_USAGE_WEBCAM
};
```

AMF_VIDEO_ENCODER_PROFILE_ENUM

Description

```
enum AMF_VIDEO_ENCODER_PROFILE_ENUM
{
    AMF_VIDEO_ENCODER_PROFILE_BASELINE = 66,
    AMF_VIDEO_ENCODER_PROFILE_MAIN = 77,
    AMF_VIDEO_ENCODER_PROFILE_HIGH = 100
};
```

Video Encoder enumerations

The header file containing the Video Encoder enumerations definition is located in `inc\amf\components\VideoEncoderVCE.h`.

AMF_VIDEO_ENCODER_SCANTYPE_ENUM

Description

```
enum AMF_VIDEO_ENCODER_SCANTYPE_ENUM
{
    AMF_VIDEO_ENCODER_SCANTYPE_PROGRESSIVE = 0,
    AMF_VIDEO_ENCODER_SCANTYPE_INTERLACED
};
```

AMF_VIDEO_ENCODER_RATE_CONTROL_METHOD_ENUM

Description

```
enum AMF_VIDEO_ENCODER_RATE_CONTROL_METHOD_ENUM
{
    AMF_VIDEO_ENCODER_RATE_CONTROL_METHOD_CONSTRAINED_QP = 0,
    AMF_VIDEO_ENCODER_RATE_CONTROL_METHOD_CBR,
    AMF_VIDEO_ENCODER_RATE_CONTROL_METHOD_PEAK_CONSTRAINED_VBR,
    AMF_VIDEO_ENCODER_RATE_CONTROL_METHOD_LATENCY_CONSTRAINED_VBR
};
```

AMF_VIDEO_ENCODER_QUALITY_PRESET_ENUM

Description

```
enum AMF_VIDEO_ENCODER_QUALITY_PRESET_ENUM
{
    AMF_VIDEO_ENCODER_QUALITY_PRESET_BALANCED = 0,
    AMF_VIDEO_ENCODER_QUALITY_PRESET_SPEED,
    AMF_VIDEO_ENCODER_QUALITY_PRESET_QUALITY
};
```

AMF_VIDEO_ENCODER_PICTURE_STRUCTURE_ENUM

Description

```
enum AMF_VIDEO_ENCODER_PICTURE_STRUCTURE_ENUM
{
    AMF_VIDEO_ENCODER_PICTURE_STRUCTURE_NONE = 0,
    AMF_VIDEO_ENCODER_PICTURE_STRUCTURE_FRAME,
    AMF_VIDEO_ENCODER_PICTURE_STRUCTURE_TOP_FIELD,
    AMF_VIDEO_ENCODER_PICTURE_STRUCTURE_BOTTOM_FIELD
};
```

Video Encoder enumerations

The header file containing the Video Encoder enumerations definition is located in `inc\amf\components\VideoEncoderVCE.h`.

AMF_VIDEO_ENCODER_PICTURE_TYPE_ENUM

Description

```
enum AMF_VIDEO_ENCODER_PICTURE_TYPE_ENUM
{
    AMF_VIDEO_ENCODER_PICTURE_TYPE_NONE = 0,
    AMF_VIDEO_ENCODER_PICTURE_TYPE_SKIP,
    AMF_VIDEO_ENCODER_PICTURE_TYPE_IDR,
    AMF_VIDEO_ENCODER_PICTURE_TYPE_I,
    AMF_VIDEO_ENCODER_PICTURE_TYPE_P,
    AMF_VIDEO_ENCODER_PICTURE_TYPE_B
};
```

AMF_VIDEO_ENCODER_OUTPUT_DATA_TYPE_ENUM

Description

```
enum AMF_VIDEO_ENCODER_OUTPUT_DATA_TYPE_ENUM
{
    AMF_VIDEO_ENCODER_OUTPUT_DATA_TYPE_IDR,
    AMF_VIDEO_ENCODER_OUTPUT_DATA_TYPE_I,
    AMF_VIDEO_ENCODER_OUTPUT_DATA_TYPE_P,
    AMF_VIDEO_ENCODER_OUTPUT_DATA_TYPE_B
};
```

Video Decoder enumerations

The header file containing the Video Decoder enumerations definition is located in `inc\amf\components\VideoDecoderUVD.h`.

AMF_VIDEO_DECODER_MODE_ENUM

Description

```
enum AMF_VIDEO_DECODER_MODE_ENUM
{
    AMF_VIDEO_DECODER_MODE_REGULAR = 0, // DPB delay is based on number of
                                        // reference frames + 1 (from SPS)

    AMF_VIDEO_DECODER_MODE_COMPLIANT, // DPB delay is based on profile - up
                                        // to 16

    AMF_VIDEO_DECODER_MODE_LOW_LATENCY, // DPB delay is 0. Expect stream with
                                        // no reordering in P-Frames or B-
                                        // Frames. B-frames can be present as
                                        // long as they do not introduce any
                                        // frame re-ordering
};
```

AMF_TIMESTAMP_MODE_ENUM

Description

```
enum AMF_TIMESTAMP_MODE_ENUM
{
    AMF_TS_PRESENTATION = 0, // default. decoder will preserve
                              // timestamps from input to output

    AMF_TS_SORT, // decoder will resort PTS list
    AMF_TS_DECODE // timestamps reflect decode order -
                  // decoder will reuse them
};
```

Video Converter enumerations

The header file containing the Video Converter enumerations definition is located in `inc\amf\components\VideoConverter.h`.

AMF_VIDEO_CONVERTER_SCALE_ENUM

Description

```
enum AMF_VIDEO_CONVERTER_SCALE_ENUM
{
    AMF_VIDEO_CONVERTER_SCALE_INVALID = -1,
    AMF_VIDEO_CONVERTER_SCALE_BILINEAR = 0,
    AMF_VIDEO_CONVERTER_SCALE_BICUBIC = 1
};
```

Video Converter enumerations

The header file containing the Video Converter enumerations definition is located in `inc\amf\components\VideoConverter.h`.

AMF_VIDEO_CONVERTER_COLOR_PROFILE_ENUM

Description

```
enum AMF_VIDEO_CONVERTER_COLOR_PROFILE_ENUM
{
    AMF_VIDEO_CONVERTER_COLOR_PROFILE_UNKNOWN = -1,
    AMF_VIDEO_CONVERTER_COLOR_PROFILE_601 = 0,
    AMF_VIDEO_CONVERTER_COLOR_PROFILE_709 = 1,
    AMF_VIDEO_CONVERTER_COLOR_PROFILE_COUNT
};
```

Acceleration enumerations

The header file containing the AMF property access enumerations definition is located in `inc\amf\components\ComponentCaps.h`.

AMF_ACCELERATION_TYPE

Description

```
enum AMF_ACCELERATION_TYPE
{
    AMF_ACCEL_NOT_SUPPORTED = -1,
    AMF_ACCEL_HARDWARE,
    AMF_ACCEL_GPU,
    AMF_ACCEL_SOFTWARE
};
```

Job Priority enumerations

The header file containing the AMF property access enumerations definition is located in `inc\amf\components\VideoEncoderVCECaps.h`.

H264EncoderJobPriority

Description

```
enum H264EncoderJobPriority
{
    AMF_H264_ENCODER_JOB_PRIORITY_NONE,
    AMF_H264_ENCODER_JOB_PRIORITY_LEVEL1,
    AMF_H264_ENCODER_JOB_PRIORITY_LEVEL2
};
```

Property Access enumerations

The header file containing the AMF property access enumerations definition is located in `inc\amf\core\PropertyStorageEx.h`.

AMF_PROPERTY_ACCESS_TYPE

Description

```
enum AMF_PROPERTY_ACCESS_TYPE
{
    AMF_PROPERTY_ACCESS_PRIVATE = 0,
    AMF_PROPERTY_ACCESS_READ = 0x1,
    AMF_PROPERTY_ACCESS_WRITE = 0x2,
    AMF_PROPERTY_ACCESS_READ_WRITE = (AMF_PROPERTY_ACCESS_READ |
    AMF_PROPERTY_ACCESS_WRITE),
    AMF_PROPERTY_ACCESS_WRITE_RUNTIME = 0x4,
    AMF_PROPERTY_ACCESS_FULL = 0xFF,
};
```

Variant-specific enumerations

The header file containing the AMF variant type enumerations definition is located in `inc\amf\core\Variant.h`.

AMF_VARIANT_TYPE

Description

```
enum AMF_VARIANT_TYPE
{
    AMF_VARIANT_EMPTY           = 0,

    AMF_VARIANT_BOOL           = 1,
    AMF_VARIANT_INT64          = 2,
    AMF_VARIANT_DOUBLE         = 3,

    AMF_VARIANT_RECT           = 4,
    AMF_VARIANT_SIZE           = 5,
    AMF_VARIANT_POINT          = 6,
    AMF_VARIANT_RATE           = 7,
    AMF_VARIANT_RATIO          = 8,
    AMF_VARIANT_COLOR          = 9,

    AMF_VARIANT_STRING         = 10, // value is char*
    AMF_VARIANT_WSTRING        = 11, // value is wchar*
    AMF_VARIANT_INTERFACE      = 12, // value is AMFInterface*
};
```

Chapter 4

Structure Definitions

AMFRect

Description

Structure containing the coordinates for a rectangular object as well as functions to return the width and height.

The header file containing the structure definition is located in `inc\amf\core\Platform.h`.

```
struct AMFRect
{
    amf_int32 left;
    amf_int32 top;
    amf_int32 right;
    amf_int32 bottom;

    bool operator==(const AMFRect& other) const
    {
        return left == other.left &&
            top == other.top &&
            right == other.right &&
            bottom == other.bottom;
    }
    amf_int32 Width() const { return right - left; }
    amf_int32 Height() const { return bottom - top; }
};

//left -> left coordinate for the rectangular object
//top -> top coordinate for the rectangular object
//right -> right coordinate for the rectangular object
//bottom -> bottom coordinate for the rectangular object
//Width -> Returns width of the rectangular object
//Height -> Returns height of the rectangular object
```

AMFSize

Description

```
struct AMFSize
{
    amf_int32 width;
    amf_int32 height;
    bool operator==(const AMFSize& other) const
    {
        return width == other.width && height == other.height;
    }
};

//width -> width value
//height -> height value
```

AMFPoint

Description Structure containing coordinate parameters.
The header file containing the structure definition is located in inc\amf\core\Platform.h.

```

struct AMFPoint
{
    amf_int32 x;
    amf_int32 y;
    bool operator==(const AMFPoint& other) const
    {
        return x == other.x && y == other.y;
    }
};

//X -> x coordinate
//Y -> y coordinate

```

AMFRate

Description Structure containing Frame Rate parameters.
The header file containing the structure definition is located in inc\amf\core\Platform.h.

```

struct AMFRate
{
    amf_uint32 num;
    amf_uint32 den;
    bool operator==(const AMFRate& other) const
    {
        return num==other.num && den==other.den;
    }
};

//Num -> Frame rate numerator
//Den -> Frame rate denominator

```

AMFRatio

Description Structure containing Ratio parameters.
The header file containing the structure definition is located in inc\amf\core\Platform.h.

```

struct AMFRatio
{
    amf_uint32 num;
    amf_uint32 den;
    bool operator==(const AMFRatio& other) const
    {
        return num==other.num && den==other.den;
    }
};

//Num -> Numerator value for a ratio
//Den -> Denominator for a ratio

```

AMFColor

Description Structure containing color parameters.
The header file containing the structure definition is located in inc\amf\core\Platform.h.

```

struct AMFColor
{
    union
    {
        struct
        {
            amf_uint8 r;
            amf_uint8 g;
            amf_uint8 b;
            amf_uint8 a;
        };
        amf_uint32 rgba;
    };
    bool operator==(const AMFColor& other) const
    {
        return r == other.r &&
            g == other.g &&
            b == other.b &&
            a == other.a;
    }
};

//r -> 8bit Red component value
//g -> 8bit Green component value
//b -> 8bit Blue component value
//a -> 8bit Alpha component value
//rgba -> 32bit RGBA value

```

AMFEnumDescriptionEntry

Description Structure defining enumeration description parameters.
The header file containing the structure definition is located in inc\amf\core\PropertyStorageEx.h..

```

struct AMFEnumDescriptionEntry
{
    amf_int          value;
    const wchar_t*  name;
};

//Value -> value of the enum
//Name -> name of the enum

```

AMFPropertyInfo*Description*

Structure containing AMF property information related members .
The header file containing the structure definition is located in
inc\amf\core\PropertyStorageEx.h.

```

struct AMFPropertyInfo
{
    const wchar_t* name;           // Name of the property

    const wchar_t* desc;           // Description
    AMF_VARIANT_TYPE type;        // Type from AMF_VARIANT_TYPE enum

    AMF_PROPERTY_CONTENT_TYPE     // Content type
    contentType;

    AMFVariantStruct              // default value
    defaultValue;

    AMFVariantStruct minValue;    // minimum value
    AMFVariantStruct maxValue;    // maximum value
    // Access type from
    //AMF_PROPERTY_ACCESS_TYPE enum
    AMF_PROPERTY_ACCESS_TYPE
    accessType;

    const
    AMFEnumDescriptionEntry*      //Pointer to enum description entry
    pEnumDescription;             // structure

    AMFPropertyInfo() :
    name(NULL),
    desc(NULL),
    type(),
    contentType(),
    defaultValue(),
    minValue(),
    maxValue(),
    accessType(AMF_PROPERTY_ACCESS_FULL),
    pEnumDescription(NULL)
    {}

    // Method to know if property can be read from or not
    bool AllowedRead() const
    {
        return (accessType & AMF_PROPERTY_ACCESS_READ) != 0;
    }

    // Method to know if property can be written to or not
    bool AllowedWrite() const
    {
        return (accessType & AMF_PROPERTY_ACCESS_WRITE) != 0;
    }

    // Method to know if property can be updated at runtime
    bool AllowedChangeInRuntime() const
    {
        return (accessType & AMF_PROPERTY_ACCESS_WRITE_RUNTIME) != 0;
    }

    virtual ~AMFPropertyInfo(){}

```

AMFPropertyInfo

```
AMFPropertyInfo(const AMFPropertyInfo& property) : name(property.name),
desc(property.desc),
type(property.type),
contentType(property.contentType),
defaultValue(property.defaultValue),
minValue(property.minValue),
maxValue(property.maxValue),
accessType(property.accessType),
pEnumDescription(property.pEnumDescription)
{}

AMFPropertyInfo& operator=(const AMFPropertyInfo& property)
{
    desc = property.desc;
    type = property.type;
    contentType = property.contentType;
    defaultValue = property.defaultValue;
    minValue = property.minValue;
    maxValue = property.maxValue;
    accessType = property.accessType;
    pEnumDescription = property.pEnumDescription;

    return *this;
}
};
```

AMFVariantStruct*Description*

The structure containing AMF variant related members.
The header file containing the structure definition is located in inc\amf\core\Variant.h.

```

struct AMFVariantStruct
{
    AMF_VARIANT_TYPE    type;
    union
    {
        amf_bool        boolValue;
        amf_int64       int64Value;
        amf_double      doubleValue;
        char*           stringValue;
        wchar_t*        wstringValue;
        AMFInterface*  pInterface;
        AMFRect         rectValue;
        AMFSize         sizeValue;
        AMFPoint        pointValue;
        AMFRate         rateValue;
        AMFRatio        ratioValue;
        AMFColor        colorValue;
    };
};

//boolValue -> represents a bool value
//int64Value -> represents 64bit integer value
//doubleValue -> represents double value
//stringValue -> represents a string value
//wstringValue -> represents a wide string value
//pInterface -> represents pointer to AMFInterface
//rectValue -> parameter of type AMFRect to hold
//coordinates of a rectangular object
//sizeValue -> parameter of type AMFSize to hold
width & height parameters

//pointValue -> parameter of type AMFPoint to hold
x, y coordinates

//rateValue -> parameter of type AMFRate to hold
rate control values

//ratioValue -> parameter of type AMFRatio to hold
ratio values

//colorValue -> parameter of type AMFColor to hold
rgba color component values

```

Chapter 5

Using the AMD AMF API

The typical workflow for an application using the AMD AMF API to accelerate Video Encoding using the AMD hardware Video Coding Engine (VCE) is as follows:

- Create and initialize context
 - Allocate context via factory function `AMFCreateContext()`
 - Set or initialize device (Direct X, OpenCL, OpenGL). E.g. DX11 initializes using `AMFContext::InitDX11()`
- Create component via factory function `AMFCreateComponent()`
- Initialize encoder component
 - Set all optional properties on component
`AMFPropertyStorage::SetProperty()`
 - Initialize the component by `AMFComponent::Init()`
- Create input data object
 - Allocate input surface with attached
(`AMFContext::CreateSurfaceFrom<>`) or allocated internally
data (`AMFContext::AllocSurface`)
 - Copy input data to input data object using native data-access
functionality: `AMFSurface::GetPlane()`,
`AMFPlane::GetNative()`
- Submit data object to encoder
 - Set additional parameters on the data object (e.g. some application
ID if needed) using `AMFPropertyStorage::SetProperty()`
 - Submits data to component by `AMFComponent::SubmitInput()`
- Queries for results (likely in a separate thread) by
`AMFComponent::QueryOutput`
- At the end of the file execute `drain` to force the component to return all
accumulated frames: `AMFComponent::Drain()`
- Checks for EOF error returning from `QueryResult()` to detect end of drain
- Terminate component and context
 - Terminate component and release all internal resources by
`AMFComponent::Terminate()`
 - Terminate context by `AMFContext::Terminate()`

- Release the AMFContext and AMFComponent pointers

The following figure depicts these steps.

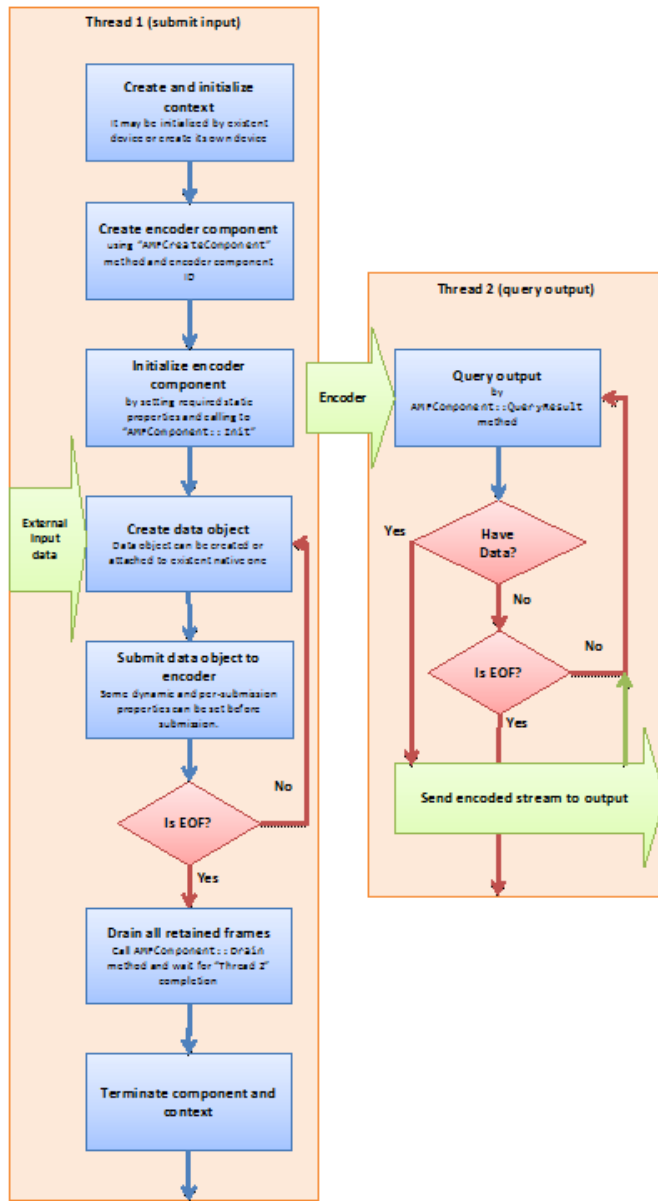


Figure 5.1 Typical workflow for an application using the AMD AMF API

The AMF Video Encoder component is based on the standard AMF interfaces, as shown in the following figure:

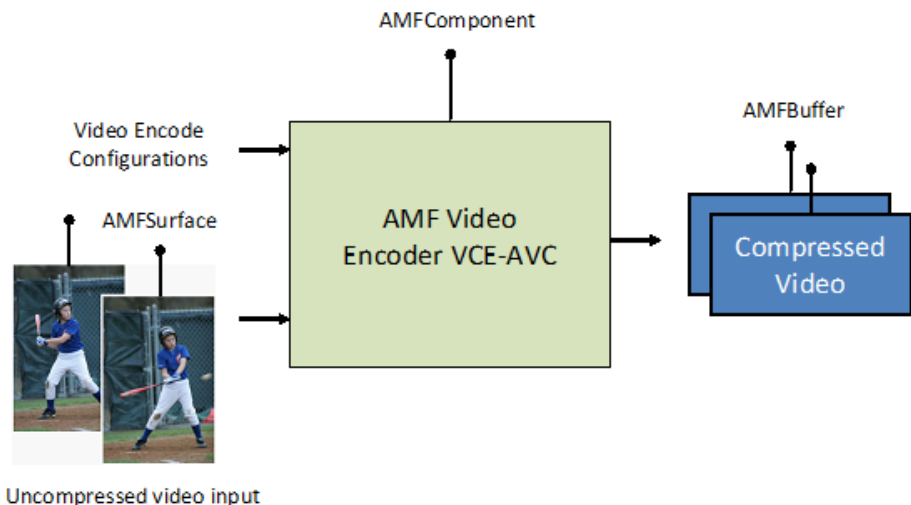


Figure 5.2 AMF Interfaces used in the AMD Video Encoder

The AMF Video Encoder uses these interfaces in the following manner:

- The Video Encoder object implements the `AMFComponent` interface.
- The Video Encoder object can be created by the `AMFCreateComponent` object using `AMFVideoEncoderHW_AVC`.
- Input frames are represented by an implementation of the `AMFSurface` interface
- Compressed output bit stream buffers implement the `AMFBuffer` interface.

5.1 Device Selection (DirectX9/Direct11/OpenGL/OpenCL)

The `AMFContext` allows interoperability with different hardware acceleration frameworks. It can create a new device context or be attached to an existing one to provide efficient interoperability with other portions of the application pipeline.

Once the `AMFContext` object is initialized with the appropriate device, the `AMFCreateComponent` function can be called to create an instance of the encoder. Since the `AMFContext` interface is derived from the `AMFPropertyStorage` interface, it provides support to query read-only properties of the selected device.

To initialize the device, the following components must be initialized:

- The OpenCL engine must be initialized (`AMFContext::InitOpenCL`) before using the AMF Video Encoder.
- The DirectX (9 or 11) engine must be initialized (`AMFContext::InitDX9` or `AMFContext::InitDX11`) when OpenGL or OpenCL is used in the AMF Video Encoder.

- DirectX 9 must be initialized when DirectX 11.0 is used.

5.2 Video Encoder I/O

5.2.1 Video Encoder Input

Input frames are represented by the `AMFSurface` interface when submitted to the encoder component. The encoder will implicitly control various aspects of the encoding job (e.g., resolution) based on the input frame properties. The encoder will automatically detect any dynamic resolution changes and apply them to the encoded bit stream.

5.2.2 Video Encoder Output

The following figure illustrates how to use the encoder component's `QueryResult()` call to retrieve the encoded video bitstream buffers:

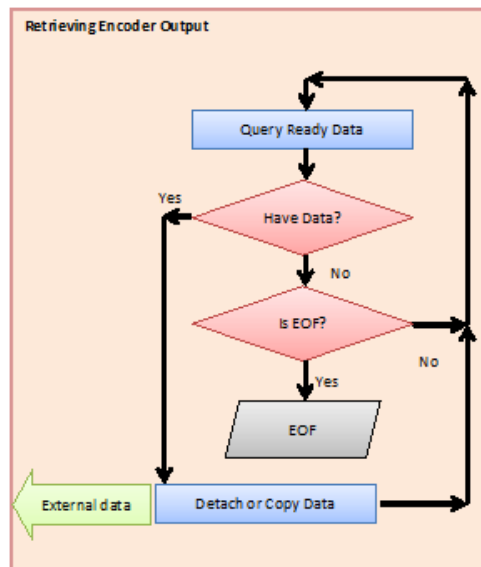


Figure 5.3 Retrieving the encoded video bitstream buffers

This call is non-blocking, which allows it to be used either in the input data thread, or in a separate output thread. The call will return immediately, even if there is no encoded bitstream data available to pick-up.

Note: In the event that B-pictures are enabled, the encoder may require multiple input frames before outputting encoded frames. The `Drain()` call can be used to flush the encoder's internal queue of input and output frames. B-pictures are bi-predicted frames which can use both previous and forward frames for data reference to get the highest amount of data compression.

Chapter 6

Pipeline Framework

Chapter 5 described how to use the AMF APIs to create, initialize, execute and terminate individual processing elements for any use-case. This chapter focusses more on encapsulating a wrapper or framework over the AMF APIs to ease the use of the APIs and to provide a starting point for developers working on AMF.

6.1 Framework

The framework uses a pipeline-based architecture. To conceptualize the framework, review the simple encode pipeline provided in the SimpleEncode sample and depicted below:

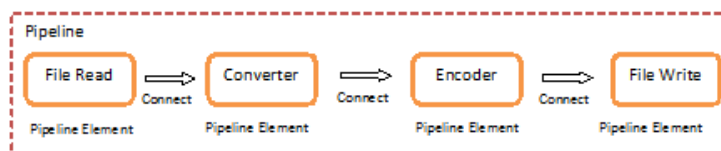


Figure 6.1 Pipeline Framework and Elements

As observed in the above figure, each AMF Component in a use-case is called "Pipeline Element" and a collection of Pipeline Elements is called "Pipeline".

The Pipeline Elements are "connected" together to enable data flow between the pipeline elements.

6.2 Class definitions of pipeline elements

Pipeline Element

Description The PipelineElement class is the base class describing methods to submit input and receive output from AMF Component (PipelineElement).

The header file containing the interface definition located in samples\amf\common\inc\PipelineElement.h.

The source file containing the definition of these interface is located in samples\amf\common\src\PipelineElement.cpp.

```
class PipelineElement
{
public:
// Get the number of slots supported at the input of the element
virtual amf_int32 GetInputSlotCount() { return 1; }

// The number of slots supported at the output of the element
virtual amf_int32 GetOutputSlotCount() { return 1; }

// Method to feed data to the pipeline element
virtual AMF_RESULT SubmitInput(amf::AMFData* pData, amf_int32 slot) {
return SubmitInput(pData); }
virtual AMF_RESULT SubmitInput(amf::AMFData* pData) { return
AMF_NOT_SUPPORTED; }

// Method to receive output from the pipeline element
virtual AMF_RESULT QueryOutput(amf::AMFData** ppData, amf_int32 slot) {
return QueryOutput(ppData); }
virtual AMF_RESULT QueryOutput(amf::AMFData** ppData) { return
AMF_NOT_SUPPORTED; }

// Method to drain the pipeline element
virtual AMF_RESULT Drain() { return AMF_NOT_SUPPORTED; }

// Method to print the execution result
virtual std::wstring GetDisplayResult() { return std::wstring(); }

virtual ~PipelineElement(){}
protected:
PipelineElement():m_host(0){}

Pipeline* m_host;
};
```

Pipeline

Description

The Pipeline class is the base class from which the other use-case specific pipelines such as `EncodePipeline`, `DecodePipeline`, `TranscodePipeline` are derived. It contains methods to connect individual pipeline element and form a use-case specific Pipeline, Start/Stop processing data in a Pipeline.

The header file containing the interface definition located in `samples\amf\common\inc\Pipeline.h`.

The source file containing implementation of the interface is location in `samples\amf\common\src\Pipeline.cpp`.

```
class Pipeline
{
friend class PipelineConnector;
typedef std::shared_ptr<PipelineConnector> PipelineConnectorPtr;

public:
Pipeline();
virtual ~Pipeline();

// Method to connect one pipeline element to another
AMF_RESULT Connect(
PipelineElementPtr pElement,
amf_int32 queueSize,
bool synchronized = false);

AMF_RESULT Connect(
PipelineElementPtr pElement,
amf_int32 slot,
PipelineElementPtr upstreamElement, amf_int32 upstreamSlot,
amf_int32 queueSize,
bool synchronized = false);

// Start a Pipeline
virtual AMF_RESULT Start();

// Stop a Pipeline
virtual AMF_RESULT Stop();

// Get state of Pipeline: NotReady, Ready, Running, EOF
PipelineState GetState();

// Performance and Display related methods
virtual void DisplayResult();
double GetFPS();
double GetProcessingTime();
amf_int64 GetNumberOfProcessedFrames();

private:
// This method is executed when EOF is encountered
void OnEof();

amf_int64 m_startTime;
amf_int64 m_stopTime;

typedef std::vector<PipelineConnectorPtr> ConnectorList;
ConnectorList m_connectors;
PipelineState m_state;
AMFCriticalSection m_cs;
};
```

6.3 Application Workflow

The following is the typical workflow of an application that uses the AMF APIs:

- Create a use-case specific pipeline derived from the `Pipeline` class
- Initialize the Pipeline
- Start/run the Pipeline
- Terminate the Pipeline

6.3.1 Initialize the pipeline

- Create AMF Context: `AMFCreateContext(&m_pContext);`
- Initialize the DirectX Device based on the user specified engine memory type. If engine Memory Type == DX9, initialize the appropriate DirectX 9 device (adapterID)

```
res = m_deviceDX9.Init(true, adapterID, false, 1, 1);
res = m_pContext->InitDX9(m_deviceDX9.GetDevice());
```

The above functions are defined in:

```
// Header file: samples\amf\common\inc\DeviceDX9.h
// Source file: samples\amf\common\src\DeviceDX9.cpp
```

If engine memory type == DX11, initialize the appropriate DirectX 11 device (adapter ID)

```
res = m_deviceDX11.Init(adapterID, false);
res = m_pContext->InitDX11(m_deviceDX11.GetDevice());
```

The above functions are defined in:

```
// Header file: samples\amf\common\inc\DeviceDX11.h
// Source file: samples\amf\common\src\DeviceDX11.cpp.
```

- Create and initialize the the individual `PipelineElements` of the Pipeline.
 - Simple Encoder Pipeline:**

For example, in the case of Simple Encoder, the pipeline is:
Raw File Read ->Encoder ->Encoded Stream File Write.

The **Raw File Reader:**

- Initializes the raw file reader pipeline element.
- Reads uncompressed data
- Supported formats: ARGB, BGRA, RGBA, NV12, YV12, YUV420P
- Header file: `samples\amf\common\inc\RawStreamReader.h`
Source file: `samples\amf\common\inc\RawStreamReader.cpp`
- In addition to the `init` method, also implements component-specific `QueryOutput()` method.

The Encoder:

- Corresponds to the Encoder pipeline element.
- The Encoder engine depends on how context is initialized (DX9, DX9Ex or DX11).
- The native input memory type is the same as engine type. Natively supports only NV12 as input (both DX9/DX11 case). If passed any other format BGRA, AGRA, RGBA, YV12, YUV420P, it will be converted by the internal converter before being passed along to the Encoder block.
- The incoming surface format must be specified in `m_pEncoder->Init`.
- The Encoder output buffers are only in HOST memory because they are compressed.

Encoder input buffers are AMF Surfaces. AMF supports several video memory types for 2D surfaces: DX9, DX9Ex, DX11, OpenCL, OpenGL, Host.

AMF provides conversion between these types using two methods, `AMFData::Convert()` and `AMFData::Duplicate()`. These conversions will try to use interop if possible. If interop is not available AMF will still do conversion through HOST memory.

The Compressed Stream Writer:

- Initializes the encoded stream writer pipeline element.
- Header file: `samples\amf\common\inc\PlatformWindows.h`.
Source file: `samples\amf\common\src\PlatformWindows.h`.

Simple Decoder Pipeline:

Similarly, in the Simple Decoder sample, the pipeline is:
Encoded Stream File Parser -> Decoder -> Converter -> Raw Data Writer.

The Compressed Stream Parser:

- Initializes the Parser.
- Currently supports H.264 Elementary stream parsing.
- Header files: `samples\amf\common\inc\BitStreamParser.h` and `BitStreamParserH264.h`.
Source files:
`samples\amf\common\src\BitStreamParser.cpp` and `BitStreamParserH264.cpp`.

The Decoder:

- Corresponds to the Decoder pipeline element.
- The Decoder engine depends on how context is initialized (DX9, DX9Ex or DX11).
- The Output memory type is the same as engine type. Natively (without internal conversion) outputs only NV12 (both DX9/DX11 case).

Use internal converter to receive DX9: (BGRA/NV12); DX11: (BGRA/RGBA/NV12).

Outgoing surface format must be specified in `m_pDecoder->Init`.

Decoder input buffers are only in HOST memory because they are compressed.

Decoder output buffers are AMF Surfaces. AMF supports several video memory types for 2D surfaces: DX9, DX9Ex, DX11, OpenCL, OpenGL, Host.

AMF provides conversion between these types using two methods, `AMFData::Convert()` and `AMFData::Duplicate()`. These conversions will try to use interop if possible. If interop is not available AMF will still do conversion through HOST memory.

The Converter:

- Component to Color convert, scale input content.
- Supported Native memory types - DX9, DX9Ex, DX11 and OpenCL. Converter Engine depends on output format (`AMF_VIDEO_CONVERTER_MEMORY_TYPE`). Can be HOST, OpenCL or internal GPU processing>

In case of output memory type HOST, engine will be CPU (HOST).
In case of output memory type DX9/DX11 and no OpenGL/OpenCL device in context- engine will be internal GPU processing.

In case of output memory type DX9/DX11 and OpenGL/OpenCL device in context - engine will be OpenCL

Internally, the converter uses one of two technologies: Internal GPU Processing or OpenCL. If the application explicitly calls `AMFContext::InitOpenCL()`, the converter will use OpenCL. If not, it will invoke Internal GPU Processing.

- Supports input formats: NV12, BGRA, AGRA, RGBA, YV12, YUV420P.
Supports output formats: NV12, BGRA, AGRA, RGBA, YV12, YUV420P, but limited by memory type:

DX9: (BGRA/NV12)
 DX11: (BGRA/RGBA/NV12)
 OpenGL (BGRA/RGBA)
 OpenCL (NV12, BGRA, AGRA, RGBA, YV12, YUV420P).

- Converter input and output buffers are AMF Surfaces.
 AMF supports several video memory types for 2D surfaces: DX9, DX9Ex, DX11, OpenCL, OpenGL, Host.

AMF provides conversion between these types using two methods, `AMFData::Convert()` and `AMFData::Duplicate()`. These conversions will try to use interop if possible. If interop is not available AMF will still do conversion through HOST memory.

The Raw Data Writer:

- Initializes the raw decoded output.
- Writes the uncompressed, decoded data.
- **Header file:** `samples\amf\common\inc\RawStreamWriter.h`
Source file:
`samples\amf\common\inc\RawStreamWriter.cpp`.
- In addition to the `init` method, also implements component specific `SubmitInput()` method.

After the individual pipeline elements (AMF Components) are created and initialized, they must be connected to form a Pipeline.

In the case of **Simple Encoder**, the connect pipeline would look like this:

```
Connect(m_pReader, 4);
Connect(PipelineElementPtr(new PipelineElementEncoder(m_pEncoder,
pParams,
frameParameterFreq, dynamicParameterFreq)), 10);
Connect(m_pStreamWriter, 5);
```

In the case of **Simple Decoder**, the connect pipeline would look like this:

```
Connect(m_pParser, 10);
Connect(PipelineElementPtr(new
PipelineElementAMFComponent(m_pDecoder)), 4);
Connect(PipelineElementPtr(new
PipelineElementAMFComponent(m_pConverter)), 4);
Connect(m_pFramesConsumer, 4);
```

The Pipeline Class defined earlier defines the 'connect' routine, wherein the first parameter is the 'PipelineElement' and the second is the 'queue' size.

For each element in the Pipeline, an object of type `PipelineConnector` is created to connect two pipelines. The output of one pipeline element is connected to the input of another element based on the slot numbers.

The class definition of the `PipelineConnector` class is available in `Pipeline.cpp`.

PipelineConnector

Description The `PipelineConnector` class is the class defining the connection between the output of one element to the input of another element.

The source file containing the definition of these interfaces is located in `samples\amf\common\src\Pipeline.cpp`.

```
class PipelineConnector
{
    friend class Pipeline;
    friend class InputSlot;
    friend class OutputSlot;
protected:

public:
    PipelineConnector(Pipeline *host, PipelineElementPtr
element);
    virtual ~PipelineConnector();

    // Method to start and stop data flow
    void Start();
    void Stop();
    bool StopRequested() {return m_bStop;}

    void NotLast() {m_bLast = false;}
    void NotPush() {m_bPush = false;}

    void OnEof();

    // a-sync operations from threads
    AMF_RESULT Poll(amf_int32 slot);
    AMF_RESULT PollAll(bool bEof);

    // methods of add input and output slots
    void AddInputSlot(InputSlotPtr pSlot);
    void AddOutputSlot(OutputSlotPtr pSlot);

    // Methods of get the number of frames processed
    amf_int64 GetSubmitFramesProcessed() {return
m_iSubmitFramesProcessed;}
    amf_int64 GetPollFramesProcessed() {return
m_iPollFramesProcessed;}

protected:
    Pipeline*           m_pHost;
    PipelineElementPtr m_pElement;
    bool                m_bPush;
    bool                m_bLast;
    bool                m_bStop;
    amf_int64           m_iSubmitFramesProcessed;
    amf_int64           m_iPollFramesProcessed;

    std::vector<InputSlotPtr> m_InputSlots;
    std::vector<OutputSlotPtr> m_OutputSlots;
};
```

Slot

Description

The base class from which the input and output slot classes are derived.

The source file containing the definition of these interfaces is located in `samples\amf\common\src\Pipeline.cpp`.

```
class Slot : public AMFThread
{
public:
    bool                m_bThread;
    PipelineConnector  *m_pConnector;
    amf_int32           m_iThisSlot;
    AMFPreciseWaiter   m_waiter;

    Slot(bool bThread, PipelineConnector *connector,
          amf_int32 thisSlot);
    virtual ~Slot() {}

    void Stop();
    virtual bool StopRequested();
};
```

InputSlot

Description

Defines the connection at the input of a component. Since data is fed to the input of the component, this class implements the 'submitInput' method.

The source file containing the definition of these interfaces is located in `samples\amf\common\src\Pipeline.cpp`.

```
class InputSlot : public Slot
{
public:
    OutputSlot          *m_pUpstreamOutputSlot;

    InputSlot(bool bThread, PipelineConnector *connector,
              amf_int32 thisSlot);
    virtual ~InputSlot() {}

    // pull
    virtual void Run();
    AMF_RESULT Drain();
    AMF_RESULT SubmitInput(amf::AMFData* pData, amf_ulong
                           ulTimeout, bool poll);
};
typedef std::shared_ptr<InputSlot> InputSlotPtr;
```

OutputSlot

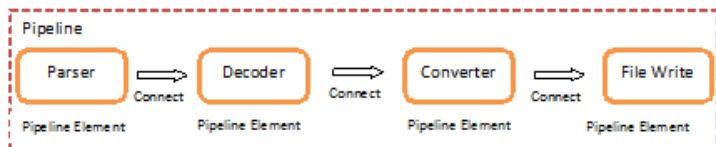
<i>Description</i>	<p>Defines the connection at the output of a component. Since data is received from the output of the component, this class implements the 'queryOutput' method.</p> <p>The source file containing the definition of these interfaces is located in <code>samples\amf\common\src\Pipeline.cpp</code>.</p> <pre> class OutputSlot : public Slot { public: DataQueue m_dataQueue; InputSlot *m_pDownstreamInputSlot; OutputSlot(bool bThread, PipelineConnector *connector, amf_int32 thisSlot, amf_int32 queueSize); virtual ~OutputSlot(){} virtual void Run(); AMF_RESULT QueryOutput(amf::AMFData** ppData, amf_ulong ulTimeout); AMF_RESULT Poll(bool bEof); }; </pre>
--------------------	---

In the example of the encoder and decoder pipelines, the connection between the components would look like this:

Encoder Pipeline:



Decoder Pipeline:



`PipelineConnector` connects the Output of one pipeline element to the Input of another pipeline element. The first element in the Pipeline will have only an Output Slot and the last element in the Pipeline will have only an Input Slot.

6.3.2 Start/Run the pipeline

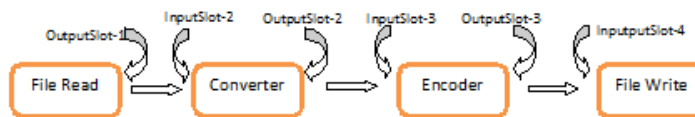
- Executes the pipeline.

Invokes `PipelineConnector::Start()` for each slot (Input and Output) of the Pipeline Elements. This creates a new thread for each slot of the processing elements.

Each `InputSlot` and `OutputSlot` would be separate threads.

So the entire Pipeline is created in one thread, whereas the slot processing (input or output) are in separate threads.

The following example shows how the individual thread would operate in case of the Simple Encoder.



- Next, the `Run()` method on each of the slot threads is invoked.
`InputSlot::Run()`
`OutputSlot::Run()`
 These methods are defined in `Pipeline.cpp`.
- `InputSlot::Run()`
 This method checks whether there is any data buffer available in the queue. If TRUE, invoke `submitInput` at the input of the component: `m_pElement->SubmitInput`.
- `OutputSlot::Run()`
 Poll the output for any data `m_pElement->QueryOutput`. If data is generated, add it to the queue.

6.3.3 Terminate the pipeline

This releases all components and Devices in the Pipeline.