Advanced Micro Devices

Advanced Media Framework API Reference

Disclaimer

Copyright Notice

MIT license

Contents

1 Introduction

AMF SDK 1.3 is intended to assist Independent Software Vendors (ISV) in development of multimedia applications using AMD GPU and APU devices when the use of Microsoft Media Foundation Framework is undesireable.

AMF is a light-weight, portable multimedia framework that abstracts away most of the platform and API-specific details and allows for easy implementation of multimedia applications using a variety of technologies, such as DirectX9, DirectX11, DirectX11.1, OpenGL, OpenCL and facilitates an efficient interop between them.

The AMF framework is compatible with most recent Radeon GPUs starting with the Southern Islands family and APUs of the Kabini, Kaveri, Carrizo families and newer.

The AMF run-time is included in the latest Windows Crimson driver 16.7.2.

2 Definitions, Acronyms and Abbreviations

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	AMD C++ API created to build flexible pipelines
SI	Southern Islands	GPU family
WinRT	Windows Runtime	Short name for Windows Store Application API

AMF API

2.1 Elementary Data Types

Elementary data types and AMF types are defined to make code potentially portable to other OSs. Detailed list of Elementary Data types is available in *public/include/core/Platform.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;
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;
#define AMF_STD_CALL            __stdcall
#define AMF_CDECL_CALL          __cdecl
#define AMF_FAST_CALL           __fastcall
#define AMF_INLINE              inline
#define AMF_FORCEINLINE         __forceinline
```

2.2 Core Interfaces and Classes

2.2.1 AMF Run-time Initialization

The *AMFFactory* interface is the entry point for the AMF run-time. It is used to create other AMF objects.

The AMF run-time is supplied as part of the Windows driver installation. The AMF run-time DLL should be loaded dynamically using the *LoadLibraryW* Win32 function. The name of the DLL is defined by the *AMF_DLL_NAME* macro. Always pass the *AMF_DLL_NAME* macro to *LoadLibraryW* instead of the actual DLL name to ensure code portability as the name might be defined differently depending on the platform:

```
HMODULE hAMFDll = LoadLibraryW(AMF_DLL_NAME);
```

To check the run-time version, acquire a pointer to and call the *AMFQueryVersion* function:

```
AMFQueryVersion_Fn queryVersion = (AMFQueryVersion_Fn)GetProcAddress(hAMFDll,
AMF_QUERY_VERSION_FUNCTION_NAME);
amf_uint64 version = 0;
AMF_RESULT res = queryVersion(&version);
```

Acquire a pointer to and call the initialization routine to obtain a pointer to the *AMFFactory* interface:

```
AMFInit_Fn init = GetProcAddress(hAMFDll, AMF_INIT_FUNCTION_NAME);
AMFFactory* pFactory(nullptr);
```

```
        AMF_RESULT initRes = init(version, &pFactory);
```

Include *public/include/core/Factory.h*

2.2.2 AMFInterface

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

All AMF interfaces except *AMFFactory*, *AMFTrace*, *AMFDebug* and *AMFPrograms* inherit from *AMFInterface*.

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

You should never call *delete* on any of the AMF interfaces. Instead, for reference-counted interfaces derived from AMFInterface, call *Acquire* when a new copy of the pointer pointing to an interface is created and *Release* when a pointer is destroyed. For interfaces to static objects, nothing needs to be done to manage their lifecycle.

AMF provides the *AMFInterfacePtr_T* template, which implements a “smart” pointer to an AMF interface. *AMFInterfacePtr_T* automatically increments the reference count of the object on assignment and decrements the reference count when going out of scope. Use of smart pointers is highly recommended and encouraged to avoid memory and resource leaks.

Include *public/include/core/Interface.h*

AMFInterface::Acquire

```
amf_long AMF_STD_CALL Acquire();
```

Increment the reference count on the object.

AMFInterface::Release

```
amf_long AMF_STD_CALL Release();
```

Decrement the reference count on the object.

AMFInterface::QueryInterface

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

Retrieve a pointer to the specified interface implemented by the object.

Parameter	**Description**
interfaceID [in]	The identifier of the interface being requested
ppInterface [out]	The address of a pointer variable that receives a pointer to the interface. The reference counter is incremented by 1 before being placed in *ppInterface*. Do not call *Acquire()* on *ppInterface* unless the pointer is being copied to another variable. Call *Release()* before the pointer is destroyed.
Return Value	*AMF_OK* if the interface is supported, otherwise *AMF_NO_INTERFACE*

2.2.3 AMFFactory

The AMFFactory interface is used to create AMF objects. *AMFFactory* is not derived from *AMFInterface* and is not reference-counted. Do not destroy the *AMFFactory* interface.

AMFFactory::CreateContext

```
AMF_RESULT AMF_STD_CALL CreateContext(amf::AMFContext** ppContext);
```

Create a device context object.

Parameter	Description
ppContext [out]	The address of a pointer variable that receives a pointer to the *AMFContext* interface. The reference counter is incremented by 1 before being placed in *ppInterface*. Do not call *Acquire()* on *ppInterface* unless the pointer is being copied to another variable. Call *Release()* before the pointer is destroyed.
Return Value	*AMF_OK* if the interface is supported, otherwise *AMF_NO_INTERFACE*

AMFFactory::CreateComponent

```
AMF_RESULT AMF_STD_CALL CreateComponent(amf::AMFContext* pContext, const wchar_t* id,
       amf::AMFComponent** ppComponent);
```

Create an AMF component.

Parameter	Description
pContext [in]	A pointer to the *AMFContext* interface
id [in]	The identifier of the component being requested
ppComponent [out]	The address of a pointer variable that receives a pointer to the interface. The reference counter is incremented by 1 before being placed in *ppInterface*. Do not call *Acquire()* on *ppInterface* unless the pointer is being copied to another variable. Call *Release()* before the pointer is destroyed.
Return Value	*AMF_OK* if the component has been successfully instatiated, otherwise *AMF_NOT_SUPPORTED*

AMFFactory::SetCacheFolder

```
AMF_RESULT AMF_STD_CALL SetCacheFolder(const wchar_t* path);
```

Specify a folder used as a cache for precompiled Compute kernels.

Parameter	Description
path [in]	A pointer to a wide character Unicode string containing the folder path
Return Value	*AMF_OK* on success *AMF_NOT_FOUND* when the specified folder does not exist

AMFFactory::GetCacheFolder

```
const wchar_t* AMF_STD_CALL GetCacheFolder();
```

Get the Compute kernel cache path previously set with *SetCacheFolder*.

Return Value	A constant pointer to a wide character Unicode string containing the path to the Compute kernel cache folder. The string is internally allocated, do not call *free* or *delete* on this pointer.

AMFFactory::GetDebug

```
AMF_RESULT AMF_STD_CALL GetDebug(amf::AMFDebug** ppDebug);
```

Get a pointer to the *AMFDebug* interface. The *AMFDebug* interface is not reference counted.

Parameter	**Description**
ppDebug [out]	A pointer to a memory location to receive a pointer to the *AMFDebug* interface
Return Value	*AMF_OK*

AMFFactory::GetTrace

```
AMF_RESULT AMF_STD_CALL GetTrace(amf::AMFTrace** ppTrace);
```

Get a pointer to the *AMFTrace* interface. The *AMFTrace* interface is not reference counted.

Parameter	**Description**
ppTrace [out]	A pointer to a memory location to receive a pointer to the *AMFTrace* interface
Return Value	*AMF_OK*

AMFFactory::GetPrograms

```
AMF_RESULT AMF_STD_CALL GetPrograms(amf::AMFPrograms** ppPrograms);
```

Get a pointer to the *AMFPrograms* interface. The *AMFPrograms* interface is not reference counted.

Parameter	**Description**
AMFPrograms [out]	A pointer to a memory location to receive a pointer to the *AMFPrograms* interface
Return Value	*AMF_OK*

2.2.4 AMFDebug

The *AMFDebug* interface provides access to the global debugging and performance measurement capabilities in AMF.

AMFDebug::EnablePerformanceMonitor

```
void AMF_STD_CALL EnablePerformanceMonitor(bool enable);
```

Enable or disable the AMF Performance Motinor

Parameter	**Description**
enable [in]	*true* to enable performance monitoring, *false* to disable

AMFDebug::PerformanceMonitorEnabled

```
bool AMF_STD_CALL PerformanceMonitorEnabled();
```

Check whether the AMF Performance Monitor is enabled.

Return Value	*true* when AMF Performance Monitor is enabled, *false* otherwise

AMFDebug::AssertEnable

```
void AMF_STD_CALL AssertsEnable(bool enable);
```

Enable or disable asserts in AMF objects

Parameter	Description
enable [in]	*true* to enable asserts, *false* to disable

AMFDebug::AssertsEnabled

```
bool AMF_STD_CALL AssertsEnabled();
```

Check whether asserts in AMF components are enabled.

Return Value	*true* when asserts are enabled, *false* otherwise

2.2.5 AMFTrace

The *AMFTrace* interface provides configuration facilities for AMF tracing functionality.

The object which implements the *AMFTrace* interface is not reference counted. Do not delete the object obtained from *AMFFactory::GetTrace*.

Include *public/include/core/Trace.h*.

AMFTrace::Trace
AMFTrace::TraceW

```
void AMF_STD_CALL TraceW(const wchar_t* src_path, amf_int32 line, amf_int32 level, const
       wchar_t* scope,amf_int32 countArgs, const wchar_t* format, ...);
void AMF_STD_CALL Trace(const wchar_t* src_path, amf_int32 line, amf_int32 level, const
       wchar_t* scope, const wchar_t* format, va_list* pArglist);
```

Output a trace to all registered traces. By default AMF outputs all traces to the debug output.

What is being output is controlled by trace level. Trace levels are cumulative – every subsequent level includes all messages of the previous level.

Each trace specifies a trace level it is associated with. A global trace level is set using the *SetGlobalLevel* method. AMF will output traces with all levels up to the current level.

Trace levels are defined as follows:

Level	Description
AMF_TRACE_ERROR	Error message
AMF_TRACE_WARNING	Warning message
AMF_TRACE_INFO	Info message
AMF_TRACE_DEBUG	Debug message
AMF_TRACE_TEST	Test message

Parameter	Description
src_path [in]	Name of the source file
line [in]	Linme number in the soutrce file
level [in]	Trace level
scope [in]	Message scope
countArgs [in]	Number of arguments after *format* or in *pArgList*

format [in]	A printf-like format string
pArgList [in]	A variable parameter list

AMFTrace::SetGlobalLevel

```
amf_int32 AMF_STD_CALL SetGlobalLevel(amf_int32 level);
```

Set global trace level. AMF trace will output all message with the trace level below or equal to the global trace level.

The following trace levels are accepted:

Level	Description
AMF_TRACE_ERROR	Error messages only
AMF_TRACE_WARNING	Warnings and errors
AMF_TRACE_INFO	Error, warning and info messages
AMF_TRACE_DEBUG	Error, warning, info and debug messages
AMF_TRACE_TEST	Error, warning, info, debug and test messages
AMF_TRACE_NOLOG	Turn off all messages

Parameter	**Description**
level [in]	Global trace level
Return Value	Previous trace level

AMFTrace::GetGlobalLevel

```
amf_int32 AMF_STD_CALL GetGlobalLevel();
```

Get global trace level.

Return Value	Current trace level

AMFTrace::SetWriterLevel

```
amf_int32 AMF_STD_CALL SetWriterLevel(const wchar_t* id, amf_int32 level);
```

Set trace level for a specific writer. This overrides the global trace level for the writer.

Parameter	**Description**
id [in]	Writer ID
level [in]	Trace level
Return Value	Previous trace level

AMFTrace::GetWriterLevel

```
amf_int32 AMF_STD_CALL GetWriterLevel(const wchar_t* ID);
```

Get trace level for a specific writer.

Parameter	**Description**
id [in]	Writer ID
Return Value	Current trace level

AMFTrace::SetWriterLevelForScope

```
amf_int32 AMF_STD_CALL SetWriterLevelForScope(const wchar_t* id, const wchar_t* scope,
       amf_int32 level);
```

Set trace level for a specific writer and scope. This overrides the global trace level for the writer.

Parameter	Description
id [in]	Writer ID
scope [in]	Scope
level [in]	Trace level
Return Value	Previous trace level

AMFTrace::GetWriterLevelForScope

```
amf_int32 AMF_STD_CALL GetWriterLevel(const wchar_t* id, const wchar_t* scope);
```

Get trace level for a specific writer and scope.

Parameter	Description
id [in]	Writer ID
scope [in]	Scope
Return Value	Current trace level

AMFTrace::SetPath

```
AMF_RESULT AMF_STD_CALL SetPath(const wchar_t* path);
```

Set AMF log file path.

Parameter	Description
path [in]	Full path to the AMF log file
Return Value	*AMF_OK* on success *AMF_FAIL* on failure

AMFTrace::GetPath

```
AMF_RESULT AMF_STD_CALL GetPath(wchar_t* path, amf_size* size);
```

Set AMF log file path.

Parameter	Description
path [in]	A pointer to a buffer to receive a full path to the AMF log file
size [in]	A pointer to the buffer size in bytes. Receives the actual length of the path string
Return Value	*AMF_OK* on success *AMF_FAIL* on failure

AMFTrace::Indent

```
void AMF_STD_CALL Indent(amf_int32 addIndent);
```

Add trace indentation.

The indentation value is added to the current indentation. Positive values shift output to the right, negative – to the left.

Parameter	Description
addIndent [in]	Indentation in character positions to be added

AMFTrace::GetIndentation

```
amf_int32 AMF_STD_CALL GetIndentation();
```

Get current indentation.

Return Value	Current indentation in character positions

AMFTrace::GetResultText

```
const wchar_t* AMF_STD_CALL GetResultText(AMF_RESULT res);
```

Convert AMF_RESULT to text.

Parameter	Description
addIndent [in]	Indentation in character positions to be added
Return Value	Current indentation in character positions

AMFTrace::SurfaceGetFormatName

```
const wchar_t* AMF_STD_CALL SurfaceGetFormatName(const AMF_SURFACE_FORMAT eSurfaceFormat);
```

Convert surface format to a string.

Parameter	Description
eSurfaceFormat [in]	Surface format as an enum
Return Value	Surface format as a string

AMFTrace::GetMemoryTypeName

```
const wchar_t* const AMF_STD_CALL GetMemoryTypeName(const AMF_MEMORY_TYPE memoryType);
```

Convert memory type to a string.

Parameter	Description
memoryType [in]	Memory type as an enum
Return Value	Memory type as a string

AMFTrace::RegisterWriter

```
void AMF_STD_CALL RegisterWriter(const wchar_t* writerID, AMFTraceWriter* pWriter,
       bool enable);
```

Register a custom trace writer.

Custom trace writers allow you to extend functionality of AMF trace by allowing to write to other 3rd party logs, such as other applications' log files.

Every writer must implement the *AMFTraceWriter* interface and have a unique writer ID.

Parameter	Description
writerID [in]	A unique writer ID

pWriter [in]	Pointer to the *AMFTraceWriter* interface
enable [in]	Initial state of the writer after registration

AMFTrace::UnregisterWriter

```
void AMF_STD_CALL UnregisterWriter(const wchar_t* writerID);
```

Unregister a previously registered writer.

Parameter	**Description**
writerID [in]	A unique writer ID

2.2.6 AMFRect

The *AMFRect* structure represents a rectangle defined by coordinates of its top-left and bottom-right corners.

AMFRect::left

```
amf_int32 left;
```

The X coordinate of the top-left corner

AMFRect::top

```
amf_int32 top;
```

The Y coordinate of the top-left corner

AMFRect::right

```
amf_int32 right;
```

The X coordinate of the bottom-right corner

AMFRect::bottom

```
amf_int32 bottom;
```

The Y coordinate of the bottom-right corner

AMFRect::Width

```
amf_int32 Width() const;
```

Calculate the width of a rectangle

AMFRect::Height

```
amf_int32 Height() const;
```

Calculate the height of a rectangle

AMFRect::operator==

```
bool operator==(const AMFRect& other) const;
```

Compare two *AMFRect* structures

Parameter	Description
other [in]	A reference to the *AMFRect* structure to be compared with
Return Value	`true` if structures are equal, *false* otherwise

AMFRect::operator!=

```
bool operator!=(const AMFRect& other) const;
```

Compare two *AMFRect* structures

Parameter	Description
other [in]	A reference to the *AMFRect* structure to be compared with
Return Value	`true` if structures are not equal, *false* otherwise

AMFConstructRect

```
AMFRect AMFConstructRect(amf_int32 left, amf_int32 top,
       amf_int32 right, amf_int32 bottom);
```

The initializer function for *AMFRect*.

Parameter	Description
left [in]	The X coordinate of the top-left corner
top [in]	The Y coordinate of the top-left corner
right [in]	The X coordinate of the bottom-right corner
bottom [in]	The Y coordinate of the bottom-right corner
Return Value	An instance of *AMFRect* initialized with supplied values

2.2.7 AMFSize

The *AMFSize* structure represents a size (width and height) of a two-dimensional rectangular area

AMFSize::width

```
amf_int32 width;
```

The size of the horizontal dimension of a rectangular area

AMFSize::height

```
amf_int32 height;
```

The size of the vertical dimension of a rectangular area

AMFSize::operator==

```
bool operator==(const AMFSize& other) const;
```

Compare two *AMFSize* structures

Parameter	Description
other [in]	A reference to the *AMFSize* structure to be compared with
Return Value	`true` if structures are equal, *false* otherwise

AMFSize::operator!=

```
bool operator!=(const AMFSize& other) const;
```

Compare two *AMFSize* structures

Parameter	Description
other [in]	A reference to the *AMFSize* structure to be compared with
Return Value	`true` if structures are not equal, *false* otherwise

AMFConstructSize

```
AMFSize AMFConstructSize(amf_int32 width, amf_int32 height);
```

The initializer function for *AMFRect*.

Parameter	Description
width [in]	The width of the area
height [in]	The height of the area
Return Value	An instance of *AMFSize* initialized with supplied values

2.2.8 AMFPoint

The *AMFPoint* structure represents a point in a two-dimensional space

AMFPoint::x

```
amf_int32 x;
```

The horizontal coordinate of a point

AMFPoint::y

```
amf_int32 y;
```

The vertical coordinate of a point

AMFPoint::operator==

```
bool operator==(const AMFPoint& other) const;
```

Compare two *AMFPoint* structures

Parameter	Description
other [in]	A reference to the *AMFPoint* structure to be compared with
Return Value	`true` if structures are equal, *false* otherwise

AMFPoint::operator!=

```
bool operator!=(const AMFPoint& other) const;
```

Compare two *AMFPoint* structures

Parameter	Description
other [in]	A reference to the *AMFPoint* structure to be compared with
Return Value	`true` if structures are not equal, *false* otherwise

AMFConstructPoint

```
AMFPoint AMFConstructPoint(amf_int32 x, amf_int32 y);
```

The initializer function for *AMFPoint*.

Parameter	Description
x [in]	The horizontal coordinate of a point
y [in]	The vertical coordinate of a point
Return Value	An instance of *AMFPoint* initialized with supplied values

2.2.9 AMFRate

The *AMFRate* structure represents a frame rate in the form of numerator and denominator

AMFRate::num

```
amf_int32 num;
```

The numerator

AMFRate::den

```
amf_int32 den;
```

The denominator

AMFRate::operator==

```
bool operator==(const AMFRate& other) const;
```

Compare two *AMFRate* structures

Parameter	Description
other [in]	A reference to the *AMFRate* structure to be compared with
Return Value	`true` if structures are equal, *false* otherwise

AMFRate::operator!=

```
bool operator!=(const AMFRate& other) const;
```

Compare two *AMFRate* structures

Parameter	Description
other [in]	A reference to the *AMFRate* structure to be compared with

Return Value	`true` if structures are not equal, *false* otherwise

AMFConstructRate

```
AMFRate AMFConstructRate(amf_int32 num, amf_int32 den);
```

The initializer function for *AMFRate*.

Parameter	Description
num [in]	The numerator
den [in]	The denominator
Return Value	An instance of *AMFRate* initialized with supplied values

2.2.10 AMFRatio

The *AMFRatio* structure represents an aspect ratio of a rectangular area in the form of numerator and denominator

AMFRatio::num

```
amf_int32 num;
```

The numerator

AMFRatio::den

```
amf_int32 den;
```

The denominator

AMFRatio::operator==

```
bool operator==(const AMFRatio& other) const;
```

Compare two *AMFRatio* structures

Parameter	Description
other [in]	A reference to the *AMFRatio* structure to be compared with
Return Value	`true` if structures are equal, *false* otherwise

AMFRate::operator!=

```
bool operator!=(const AMFRatio& other) const;
```

Compare two *AMFRatio* structures

Parameter	Description
other [in]	A reference to the *AMFRatio* structure to be compared with
Return Value	`true` if structures are not equal, *false* otherwise

AMFConstructRatio

```
AMFRatio AMFConstructRatio(amf_int32 num, amf_int32 den);
```

The initializer function for *AMFRatio*.

Parameter	Description
num [in]	The numerator
den [in]	The denominator
Return Value	An instance of *AMFRatio* initialized with supplied values

2.2.11 AMFColor

The *AMFColor* structure represents a 32-bit ARGB color value

AMFColor::r

```
amf_int8 r;
```

The red color component

AMFColor::g

```
amf_int8 g;
```

The green color component

AMFColor::b

```
amf_int8 b;
```

The blue color component

AMFColor::a

```
amf_int8 a;
```

The alpha component

AMFColor::rgba

```
amf_int32 rgba;
```

The composite representation (RGBA) of a color.

AMFColor::operator==

```
bool operator==(const AMFColor& other) const;
```

Compare two *AMFColor* structures

Parameter	Description
other [in]	A reference to the *AMFColor* structure to be compared with
Return Value	`true` if structures are equal, *false* otherwise

AMFColor::operator!=

```
bool operator!=(const AMFColor& other) const;
```

Compare two *AMFColor* structures

Parameter	Description
other [in]	A reference to the *AMFColor* structure to be compared with
Return Value	`true` if structures are not equal, *false* otherwise

AMFConstructColor

```
AMFColor AMFConstructColor(amf_int8 r, amf_int8 g, amf_int8 b, amf_int8 a);
```

The initializer function for *AMFRatio*.

Parameter	Description
r [in]	The red component
g [in]	The green component
b [in]	The blue component
a [in]	The alpha component
Return Value	An instance of *AMFColor* initialized with supplied values

2.2.12 AMFGuid

The *AMFGuid* structure represents a 128-bit globally unique identifier (GUID)

AMFGuid::AMFGuid

```
AMFGuid(amf_uint32 _data1, amf_uint16 _data2, amf_uint16 _data3, amf_uint8 _data41,
        amf_uint8 _data42, amf_uint8 _data43, amf_uint8 _data44, amf_uint8 _data45,
        amf_uint8 _data46, amf_uint8 _data47, amf_uint8 _data48);
```

The object's constructor.

AMFGuid::operator==

```
bool operator==(const AMFGuid& other) const;
```

Compare two *AMFGuid* structures

Parameter	Description
other [in]	A reference to the *AMFGuid* structure to be compared with
Return Value	`true` if structures are equal, *false* otherwise

AMFGuid::operator!=

```
bool operator!=(const AMFGuid& other) const;
```

Compare two *AMFGuid* structures

Parameter	Description
other [in]	A reference to the *AMFGuid* structure to be compared with
Return Value	`true` if structures are not equal, *false* otherwise

AMFCompareGUIDs

```
bool AMFCompareGUIDs(const AMFGuid& guid1, const AMFGuid& guid2);
```

The global GUID comparator function for *AMFGuid*.

Parameter	Description
guid1 [in]	The first GUID to compare
guid2 [in]	The second GUID to compare
Return Value	*true* when both GUIDs are identical, *false* otherwise

2.2.13 Variant

2.2.13.1 AMFVariantStruct

The *AMFVariantStruct* structure implements a universal typeless storage for basic types. The following types are supported (represented by the *AMF_VARIANT_TYPE* enumeration):

Type	Description
AMF_VARIANT_EMPTY	An empty variant which does not contain any value
AMF_VARIANT_BOOL	A Boolean
AMF_VARIANT_INT64	A 64-bit signed integer
AMF_VARIANT_DOUBLE	A double precision floating point
AMF_VARIANT_RECT	A rectangle represented by *AMFRect*
AMF_VARIANT_SIZE	A two-dimensional size (width and height) represented by *AMFSize*
AMF_VARIANT_POINT	A point in a two-dimensional space represented by *AMFPoint*
AMF_VARIANT_RATE	A frame rate represented by *AMFRate*
AMF_VARIANT_RATIO	An aspect ratio represented by *AMFRatio*
AMF_VARIANT_COLOR	An ARGB color represented by *AMFColor*
AMF_VARIANT_STRING	An ASCII string
AMF_VARIANT_WSTRING	A wide Unicode (UTF-16LE) string
AMF_VARIANT_INTERFACE	An interface pointer

The *AMFVariantStruct* structure provides a plain C encapsulation of a typeless variable. For C++ it is advised to use the *AMFVariant* class instead.

Initialization Functions

```
AMF_RESULT AMF_CDECL_CALL AMFVariantInit(AMFVariantStruct* pVariant);
```

Initialize a variant. This function initializes a variant structure and sets its type to *AMF_VARIANT_EMPTY*.

```
AMF_RESULT AMF_CDECL_CALL AMFVariantClear(AMFVariantStruct* pVariant);
```

Clear the variant by setting its type to *AMF_VARIANT_EMPTY*. If the variant contains a non-null pointer to an interface, the *Release()* method will be called on the interface.

Parameter	Description
_variant [in]	A pointer to an *AMFVariantStruct* structure
Return Value	*AMF_OK*

AMFVariantGetType

```
AMF_VARIANT_TYPE     AMF_STD_CALL AMFVariantGetType(const AMFVariantStruct* _variant);
AMF_VARIANT_TYPE&    AMF_STD_CALL AMFVariantGetType(AMFVariantStruct* _variant);
```

Get the type of data stored in a variant. The second version of the function returns the lvalue of the variant type.

Parameter	Description
_variant [in]	A pointer to an *AMFVariantStruct* structure
Return Value	A value/reference to a value of the *AMF_VARIANT_TYPE* type

Type Cast Functions

```
amf_bool AMF_STD_CALL AMFVariantGetBool(const AMFVariantStruct* _variant);
amf_int64 AMF_STD_CALL AMFVariantGetInt64(const AMFVariantStruct* _variant);
amf_double AMF_STD_CALL AMFVariantGetDouble(const AMFVariantStruct* _variant);
const char* AMF_STD_CALL AMFVariantGetString(const AMFVariantStruct* _variant);
const wchar_t* AMF_STD_CALL AMFVariantGetWString(const AMFVariantStruct* _variant);
const AMFInterface* AMF_STD_CALL AMFVariantGetInterface(const AMFVariantStruct* _variant);
AMFInterface* AMF_STD_CALL AMFVariantGetInterface(AMFVariantStruct* _variant);
const AMFRect& AMF_STD_CALL AMFVariantGetRect (const AMFVariantStruct* _variant);
const AMFSize& AMF_STD_CALL AMFVariantGetSize (const AMFVariantStruct* _variant);
const AMFPoint& AMF_STD_CALL AMFVariantGetPoint(const AMFVariantStruct* _variant);
const AMFRate& AMF_STD_CALL AMFVariantGetRate (const AMFVariantStruct* _variant);
const AMFRatio& AMF_STD_CALL AMFVariantGetRatio(const AMFVariantStruct* _variant);
const AMFColor& AMF_STD_CALL AMFVariantGetColor(const AMFVariantStruct* _variant);
```

Cast a variant to a basic type.

Parameter	Description
_variant [in]	A pointer to an *AMFVariantStruct* structure
Return Value	A value/reference to a value stored in the variant

Type Assignment Functions

```
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignBool(AMFVariantStruct* pDest, bool value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignInt64(AMFVariantStruct* pDest, amf_int64 value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignDouble(AMFVariantStruct* pDest, amf_double
       value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignString(AMFVariantStruct* pDest, const char*
       value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignWString(AMFVariantStruct* pDest, const wchar_t*
       value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignInterface(AMFVariantStruct* pDest, AMFInterface*
       value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignRect(AMFVariantStruct* pDest, const AMFRect&
       value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignSize(AMFVariantStruct* pDest, const AMFSize&
       value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignPoint(AMFVariantStruct* pDest, const AMFPoint&
       value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignRate(AMFVariantStruct* pDest, const AMFRate&
       value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignRatio(AMFVariantStruct* pDest, const AMFRatio&
       value);
AMF_RESULT AMF_CDECL_CALL AMFVariantAssignColor(AMFVariantStruct* pDest, const AMFColor&
       value);
```

Assign a value of a specific type to a variant

Parameter	Description
pDest [in]	A pointer to an *AMFVariantStruct* structure
Value [in]	A value to be assigned

Return Value	A value/reference to a value stored in the variant

AMFVariantCompare

```
AMF_RESULT AMF_CDECL_CALL AMFVariantCompare(const AMFVariantStruct* pFirst,
       const AMFVariantStruct* pSecond, bool& equal);
```

Compare two variants for equality. Two variants are equal when and only when their types and their values are equal.

Parameter	**Description**
pFirst [in]	A pointer to the first *AMFVariantStruct* structure to be compared
pSecond [in]	A pointer to the second *AMFVariantStruct* structure to be compared
equal [out]	*true* when both variants are equal, *false* otherwise
Return Value	*AMF_OK* on success *AMF_INVALID_POINTER* when either *pFirst* or *pSecond* are *nullptr*

AMFVariantCopy

```
AMF_RESULT AMF_CDECL_CALL AMFVariantCopy(AMFVariantStruct* pDest,
       const AMFVariantStruct* pSrc);
```

Copy a variant. The destination will be cleared first by calling the *AMFVariantClear* function, then safely overwritten.

Parameter	**Description**
pDest [in]	A pointer to the destination *AMFVariantStruct* structure
pSrc [in]	A pointer to the source *AMFVariantStruct* structure
Return Value	*AMF_OK* on success *AMF_INVALID_POINTER* when either *pSrc* or *pDest* are *nullptr*

AMFVariantChangeType

```
AMF_RESULT AMF_CDECL_CALL AMFVariantChangeType(AMFVariantStruct* pDest,
       const AMFVariantStruct* pSrc, AMF_VARIANT_TYPE newType);
```

Copy a variant changing the type of a value stored in a variant and performing the necessary data conversion. The original variant is left unmodified.

Parameter	**Description**
pDest [in]	A pointer to the destination *AMFVariantStruct* structure
pSrc [in]	A pointer to the source *AMFVariantStruct* structure
newType [in]	A type the source variant shall be converted to
Return Value	*AMF_OK* on success *AMF_INVALID_POINTER* when either *pSrc* or *pDest* are *nullptr* *AMF_OUT_OF_MEMORY* when there is not enough free memory to perform the conversion

2.2.13.2 AMFVariant

The *AMFVariant* class is a C++ wrapper around the *AMFVariantStruct* structure and has the equivalent functionality. It is recommended that *AMFVariant* be used in C++, rather than *AMFVariantStruct*. *AMFVariant* inherits from *AMFVariantStruct*.

Contructors

```
AMFVariant();
template<typename T>
       explicit AMFVariant(const AMFInterfacePtr_T<T>& pValue);
explicit AMFVariant(amf_bool value);
explicit AMFVariant(amf_int64 value);
explicit AMFVariant(amf_uint64 value);
explicit AMFVariant(amf_int32 value);
explicit AMFVariant(amf_uint32 value);
explicit AMFVariant(amf_double value);
explicit AMFVariant(const AMFRect& value);
explicit AMFVariant(const AMFSize& value);
explicit AMFVariant(const AMFPoint& value);
explicit AMFVariant(const AMFRate& value);
explicit AMFVariant(const AMFRatio& value);
explicit AMFVariant(const AMFColor& value);
explicit AMFVariant(const char* value);
explicit AMFVariant(const wchar_t* value);
explicit AMFVariant(AMFInterface* pValue);
```

Create and initialize an instance of *AMFVariant*.

Parameter	Description
pValue, valuet [in]	An initial value to be assigned to the instance being created

Copy Constructors

```
AMFVariant(const AMFVariant& other);
explicit AMFVariant(const AMFVariantStruct& other);
explicit AMFVariant(const AMFVariantStruct* pOther);
```

Create and initialize an instance of *AMFVariant* with a value of another *AMFVariant* or *AMFVariantStruct*.

Parameter	Description
pOther, other [in]	An initial value to be assigned to the instance being created

Assignment Operators

```
AMFVariant& operator=(const AMFVariantStruct& other);
AMFVariant& operator=(const AMFVariantStruct* pOther);
AMFVariant& operator=(const AMFVariant& other);
AMFVariant& operator=(amf_bool value);
AMFVariant& operator=(amf_int64 value);
AMFVariant& operator=(amf_uint64 value);
AMFVariant& operator=(amf_int32 value);
AMFVariant& operator=(amf_uint32 value);
AMFVariant& operator=(amf_double value);
AMFVariant& operator=(const AMFRect& value);
AMFVariant& operator=(const AMFSize& value);
AMFVariant& operator=(const AMFPoint& value);
AMFVariant& operator=(const AMFRate& value);
AMFVariant& operator=(const AMFRatio& value);
AMFVariant& operator=(const AMFColor& value);
AMFVariant& operator=(const char* value);
AMFVariant& operator=(const wchar_t* value);
AMFVariant& operator=(AMFInterface* value);
template<typename T>
       AMFVariant& operator=(const AMFInterfacePtr_T<T>& value);
```

Assign a value to a variant.

Parameter	**Description**
pOther, other, value [in]	A value to be assigned

Comparison Operators

```
bool operator==(const AMFVariantStruct& other) const;
bool operator==(const AMFVariantStruct* pOther) const;
bool operator!=(const AMFVariantStruct& other) const;
bool operator!=(const AMFVariantStruct* pOther) const;
```

Compare two variants for equality. Two variants are equal when and only when their types and their values are equal.

Parameter	**Description**
pOther, other [in]	An initial value to be assigned to the instance being created
Return Value	*true* when both variants are equal, *false* otherwise for *operator==* *false* when both variants are equal, *true* otherwise for *operator!=*

AMFVariant::Empty

```
bool AMFVariant::Empty() const;
```

Check if the variant has been assigned a value.

Return Value	*true* when the variant is empty, *false* otherwise

AMFVariant::Clear

```
void Clear();
```

Clear the variant by setting its type to *AMF_VARIANT_EMPTY*. If the variant contains a non-null pointer to an interface, the *Release()* method will be called on the interface.

Explicit Type Conversions

```
amf_bool ToBool() const;
amf_int64 ToInt64() const;
amf_uint64 ToUInt64() const;
amf_int32 ToInt32() const;
amf_uint32 ToUInt32() const;
amf_double ToDouble() const;
amf_float ToFloat() const;
AMFRect ToRect() const;
AMFSize ToSize() const;
AMFPoint ToPoint() const;
AMFRate ToRate() const;
AMFRatio ToRatio() const;
AMFColor ToColor() const;
AMFInterface* ToInterface() const;
String ToString() const;
WString ToWString() const;
```

Explicitly convert a variant to a simple type.

Important Notes:

Conversion will always succeed regardless of the variant type, but the result of conversion might be meaningless.

ToInterface does not call *Acquire* on the interface being returned, leaving it up to the caller. This makes it safe to use with smart pointers.

ToString and *ToWString* return an instance of a container class of type *AMFVariant::String* and *AMFVariant::WString* respectively. These containers store a copy of the string contained in the variant and can maintain their lifecycle independently of the variant they were obtained from.

Return Value	The value of the variant cast to the corresponding type

Implicit Casts

```
operator amf_bool() const;
operator amf_int64() const;
operator amf_uint64() const;
operator amf_int32() const;
operator amf_uint32() const;
operator amf_double() const;
operator amf_float() const;
operator AMFRect () const;
operator AMFSize () const;
operator AMFPoint() const;
operator AMFRate () const;
operator AMFRatio() const;
operator AMFColor() const;
operator AMFInterface*() const;
```

Implicitly cast a variant to a simple type. While implicit casts are available for convenience, it is high recommended to use explicit type conversions instead as a safe programming practice.

Return Value	The value of the variant cast to the corresponding type

AMFVariant::ChangeType

```
void ChangeType(AMF_VARIANT_TYPE type, const AMFVariant* pSrc = nullptr);
```

Change type of a variant, optionally copying the value from another variant.

Parameter	Description
pSrc [in]	A pointer to the source *AMFVariant* object. When this parameter is set to *nullptr*, the object's own type is converted
type [in]	A type the source variant shall be converted to

AMFVariant::Attach

```
void Attach(AMFVariantStruct& variant);
```

Attach another variant to the *AMFVariant* object, transferring the ownership of the content. The original *AMFVariantStruct* is invalidated and set to *AMF_VARIANT_EMPTY*.

Parameter	Description
variant [in]	A reference to an *AMFVariantStruct* structure or an *AMFVariant* object to be attached to the current instance

AMFVariant::Detach

```
AMFVariantStruct Detach();
```

Detach and return the value of the *AMFVariant* object as another variant, transferring the ownership of the content. The original *AMFVariant* object is invalidated and set to *AMF_VARIANT_EMPTY*.

Return Value	The value of the variant

2.2.13.3 AMFVariant::String and AMFVariant::WString

The *AMFVariant::String* and *AMFVariant::WString* classes are specialized containers that encapsulate ASCII and wide-character Unicode strings in a manner that makes them safe to pass across DLL and C runtime boundaries as they encapsulate memory allocations and deallocations.

Constructors

```
String();
WString();
String(const char* str);
WString(const wchar_t* str);
String(const String& other);
String(String&& other);
WString(const WString& other);
WString(WString&& other);
```

Construct and initialize a *String/WString* object.

Parameter	Description
str, other [in]	An initial value to be assigned to the instance being created

Assignment Operators

```
String& operator=(const String& p_other);
String& operator=(String&& p_other);
WString& operator=(const WString& p_other);
WString& operator=(WString&& p_other);
```

Assign a value to a *String/WString* object.

Comparison Operators

```
bool operator==(const String& p_other) const;
bool operator==(const WString& p_other) const;
bool operator!=(const String& p_other) const;
bool operator!=(const WString& p_other) const;
```

Compare the value to another String/WString object.

Parameter	**Description**
other [in]	A reference to the object to be compared with
Return Value	*true* when both strings are equal, *false* otherwise for *operator==* *false* when both strings are equal, *true* otherwise for *operator!=*

AMFVariant::String::c_str, AMFVariant::WString::c_str

```
const char* c_str() const
const wchar_t* c_str() const
```

Get a temporary pointer to the string's internal buffer. This buffer should not be saved in any place that might outlive the *String/WString* object itself.

Return Value	A pointer to the buffer containing a null-terminated string

AMFVariant::String::size, AMFVariant::WString::size

```
size_t size() const;
```

Return the size (length) of the string in characters.

Return Value	The size of the string in characters

2.2.14 Property Storage

2.2.14.1 AMFPropertyStorage

Most objects in AMF implement the *AMFPropertyStorage* or *AMFPropertyStorageEx* interfaces. *AMFPropertyStorage* implements a property map with a string as an ID and the *AMFVariantStruct* structure as data. The default implementation is *not thread-safe.*

Include *public/include/core/PropertyStorage.h*

AMFPropertyStorage::SetProperty

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

Set a property on an object

Parameter	Description
name [in]	The name of the property to be set
value [in]	The value of the specified property
Return Value	*AMF_OK*

AMFPropertyStorage::GetProperty

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

Retrieve a property value from property storage

Parameter	Description
name [in]	The name of the property to be set
value [out]	A pointer to a location to receive the value of the specified property
Return Value	*AMF_OK* or *AMF_NOT_FOUND* when the requested property is not defined

AMFPropertyStorage::GetPropertyString

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

Retrieve the value of a property from property storage as string.

Parameter	Description
name [in]	The name of the property to be set
value [out]	A pointer to a location to receive the value of the specified property. The object receiving the value must have the assignment operator accepting a plain C string defined.
Return Value	*AMF_OK* or *AMF_NOT_FOUND* when the requested property is not defined

AMFPropertyStorage::GetPropertyWString

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

Retrieve the value of a property from property storage as string.

Parameter	Description
name [in]	The name of the property to be set
value [out]	A pointer to a location to receive the value of the specified property. The object receiving the value must have the assignment operator accepting a plain C wide string defined.

Return Value	*AMF_OK* or *AMF_NOT_FOUND* when the requested property is not defined

AMFPropertyStorage::HasProperty

```
bool AMF_STD_CALL HasProperty(const wchar_t* name) const;
```

Check if a property exists

Parameter	**Description**
name [in]	The name of the property to be checked
Return Value	*true* if the property exists, *false* otherwise.

AMFPropertyStorage::GetPropertyCount

```
amf_size AMF_STD_CALL GetPropertyCount() const;
```

Get the number of properties stored in a property storage. A property's value needs to be explicitly set with *SetProperty()*, *AddTo()* or *CopyTo()* methods to be counted.

Return Value	Total number of properties in the property storage.

AMFPropertyStorage::GetPropertyAt

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

Retrieve the value of a property at a particular location specified by an index

Parameter	**Description**
index [in]	A zero-based index of the property to be retrieved
name [out]	Property name
nameSize [in]	The size of the buffer to receive the property name in characters
pValue [out]	A pointer to a location to receive the value of the specified property

AMFPropertyStorage::Clear

```
AMF_RESULT AMF_STD_CALL Clear();
```

Remove all values from a property storage

Return Value	*AMF_OK*

AMFPropertyStorage::AddTo

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

Add all properties of the current object to another object

Parameter	**Description**
pDest [in]	Destination Object
overwrite [in]	When *true*, the property at *pDest* will be overwritten even when it exits. The original value at *pDest* will be preserved when *overwrite* is set to *false*
deep [in]	Currently ignored

Return Value	*AMF_OK*

AMFPropertyStorage::CopyTo

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

Copy all properties of the current object to another object, clearing the destination first

Parameter	**Description**
pDest [in]	Destination Object
deep [in]	Currently ignored
Return Value	*AMF_OK*

AMFPropertyStorage::AddObserver

```
void AMF_STD_CALL AddObserver(AMFPropertyStorageObserver* pObserver);
```

Add an observer object which will receive notifications when one or more properties change by calling *AMFPropertyStorageObserver::OnPropertyChanged()*.

Parameter	**Description**
pObserver [in]	Pointer to the *AMFPropertyStorageObserver* interface

AMFPropertyStorage::RemoveObserver

```
void AMF_STD_CALL RemoveObserver(AMFPropertyStorageObserver* pObserver);
```

Remove the observer previously added by *AddObserver()*

Parameter	**Description**
pObserver [in]	Pointer to the *AMFPropertyStorageObserver* interface

2.2.14.2 AMFPropertyStorageObserver

The *AMFPropertyStorageObserver* interface is used as a callback to notify other objects that one of the properties of an object has changed.

AMFPropertyStorageObserver::OnPropertyChanged

```
void AMF_STD_CALL OnPropertyChanged(const wchar_t* name);
```

This method is called when a property in the property storage changes

Parameter	**Description**
name [in]	The name of the property that has changed

2.2.14.3 AMFPropertyStorageEx

The *AMFPropertyStorageEx* interface adds property description and validation features to *AMFPropertyStorage*. *AMFPropertyStorageEx* inherits from *AMFPropertyStorage*. The default implementation is *not thread-safe.*

AMFPropertyStorageEx requires the properties to be declared before they can be used. Calling

SetProperty() on an undeclared property would result in the *AMF_NOT_FOUND* error being returned.

Include *public/include/core/PropertyStorageEx.h*

AMFPropertyStorageEx::GetPropertiesInfoCount

```
amf_size AMF_STD_CALL GetPropertiesInfoCount() const;
```

Obtain the number of declared properties that have an associated property descriptor. Unlike *AMFPropertyStorage::GetPropertiesCount()*, which returns the number of properties that have been set to a specific value, GetPropertiesInfoCount returns the number of declared properties regardless of whether their values have been set or not.

Return Value	Number of registered properties

AMFPropertyStorageEx::GetPropertyInfo

```
AMF_RESULT AMF_STD_CALL GetPropertyInfo(amf_size ind, const AMFPropertyInfo** ppInfo)
       const;
AMF_RESULT AMF_STD_CALL GetPropertyInfo(const wchar_t* name, const AMFPropertyInfo**
       ppInfo) const;
```

Retrieve a property descriptor for a specific property. A property descriptor contains various information about the property, such as name, type, range and access type.

Parameter	**Description**
ind [in]	Property index
name [in]	Property name
ppInfo [out]	Pointer to the parameter information class
Return Value	*AMF_OK* – success *AMF_NOT_FOUND* – the requested property was not found due to an invalid name or index

AMFPropertyStorageEx::ValidateProperty

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

Validate the value of a property. *ValidateProperty()* also converts the type of the supplied value to the declared type of the property when an applicable conversion is available.

Parameter	**Description**
name [in]	Name of the property
value [in]	Value of the property
pOutValidated [out]	Validated value of the property
Return Value	*AMF_OK* when the value is within the range *AMF_OUT_OF_RANGE* when the value is out of range *AMF_INVALID_POINTER* when name or *pOutValidated* is *nullptr*

2.2.14.4 AMFPropertyInfo

The *AMFPropertyInfo* structure describes various parameters of a property in property storage, such as name, type, access rights and range.

Include *public/include/core/PropertyStorageEx.h*

name

```
const wchar_t* name;
```

Contains the name of the property.

desc

```
const wchar_t* desc;
```

Contains an optional human-readable description of the property.

type

```
AMF_VARIANT_TYPE type;
```

Contains the type of the property.

contentType

```
AMF_PROPERTY_CONTENT_TYPE contentType;
```

Reserved for internal use, must be set to *AMF_PROPERTY_CONTENT_DEFAULT*.

minValue

```
AMFVariantStruct minValue;
```

Contains the minimum value of the property.

maxValue

```
AMFVariantStruct maxValue;
```

Contains the maximum value of the property.

accessType

```
AMF_PROPERTY_ACCESS_TYPE accessType;
```

Contains the property's access type. Access type can have one of the following values:

Value	Description
AMF_PROPERTY_ACCESS_PRIVATE	Property is not accessible outside of the *AMFPropertyStorageEx* object
AMF_PROPERTY_ACCESS_READ	Property is readable
AMF_PROPERTY_ACCESS_WRITE	Property is writable
AMF_PROPERTY_ACCESS_READ_WRITE	A combination of *AMF_PROPERTY_ACCESS_READ* and *AMF_PROPERTY_ACCESS_WRITE*
AMF_PROPERTY_ACCESS_WRITE_RUNTIME	Property is writable and does not require re-initialization of the component after it has been changed (specialized use)
AMF_PROPERTY_ACCESS_FULL	All access is allowed, re-initialization is not required

pEnumDescription

```
const AMFEnumDescriptionEntry*  pEnumDescription;
```

A pointer to the array of *AMFEnumDescriptionEntry* structures describing an enumeration. The *AMFEnumDescriptionEntry* structure is defined as follows:

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

Field	Description
value	An integer value of the enumeration entry
name	A wide-character string containing the name of the enumeration entry

2.3 Memory Objects

2.3.1 AMFData

The *AMFData* interface abstracts memory objects located in CPU and GPU memory providing a cross-platform access to them. It serves as a base class for other interfaces used to access specific memory objects.

AMFData inherits from *AMFPropertyStorage*. AMFData objects are generally not thread-safe.

Include *public/include/core/Data.h*

AMFData::GetDataType

```
AMF_DATA_TYPE AMF_STD_CALL GetDataType();
```

Obtain the type of data in the memory block.

Data type is defined by the *AMF_DATA_TYPE* enumeration and can have one of the following values:

Value	Description
AMF_DATA_BUFFER	data is a single-dimensional general purpose buffer in system (host) memory containing any unstructured
AMF_DATA_SURFACE	data is a graphical buffer, such as a DX9 surface, a DX11 texture, an OpenGL surface, etc.
AMF_DATA_AUDIO_BUFFER	data contains an audio buffer
AMF_DATA_USER	user data

Return Value A value of *AMF_DATA_TYPE* representing the type of data

AMFData::GetMemoryType

```
AMF_MEMORY_TYPE AMF_STD_CALL GetMemoryType();
```

Obtain the type of memory the buffer is stored in.

Memory type is defined by the *AMF_MEMORY_TYPE* enumeration and can have one of the following values:

Value	Description

AMF_MEMORY_UNKNOWN	memory type is not set
AMF_MEMORY_HOST	buffer is located in host (CPU) memory
AMF_MEMORY_DX9	buffer is a DX9 surface in GPU memory
AMF_MEMORY_DX11	buffer is a DX11 texture in GPU memory
AMF_MEMORY_OPENCL	buffer is an OpenCL surface in GPU memory
AMF_MEMORY_OPENGL	buffer is an OpenGL surface in GPU memory
AMF_MEMORY_XV	buffer is an XV surface in GPU memory (Linux only)
AMF_MEMORY_GRALLOC	buffer is a GrAlloc block in GPU memory (Android only)
AMF_MEMORY_COMPUTE_FOR_DX9	buffer is a DX9 DirectCompute block in GPU memory
AMF_MEMORY_COMPUTE_FOR_DX11	buffer is a DX11 DirectCompute block in GPU memory

Return Value	A value of the AMF_MEMORY_TYPE type representing the memory type of the memory block

AMFData::Duplicate

```
AMF_RESULT AMF_STD_CALL Duplicate(AMF_MEMORY_TYPE type, AMFData** ppData);
```

Duplicate a memory block to another memory block of a different memory type, copying its content if necessary.

Parameter	**Description**
type [in]	Memory type to convert to
ppData [in]	A location to receive a pointer to the newly created object
Return Value	*AMF_OK* on success *AMF_INVALID_POINTER* when *ppData* is *nullptr* *AMF_INVALID_FORMAT* when *type* is invalid *AMF_NOT_SUPPORTED* when conversion is not supported on the current platform

AMFData::Convert

```
AMF_RESULT AMF_STD_CALL Convert(AMF_MEMORY_TYPE type);
```

Convert the current *AMFData* object to a different memory type, transferring the content to a new memory location when necessary even if it involves a copy to the host memory and back.

Parameter	**Description**
type [in]	Memory type to convert to
Return Value	*AMF_OK* on success *AMF_INVALID_FORMAT* when *type* is invalid *AMF_NOT_SUPPORTED* when conversion is not supported on the current platform

AMFData::Interop

```
AMF_RESULT AMF_STD_CALL Interop(AMF_MEMORY_TYPE type);
```

Convert the current *AMFData* object to a different memory type. Unlike *AMFData::Convert*, *AMFData::Interop* will fail when conversion requires content transfer through system memory.

Parameter	**Description**
type [in]	Memory type to convert to

Return Value	*AMF_OK* on success *AMF_INVALID_FORMAT* when *type* is invalid *AMF_NOT_SUPPORTED* when conversion is not supported on the current platform

AMFData::IsReusable

```
bool AMF_STD_CALL IsReusable();
```

Check if the data object is reusable, i.e. created by AMF rather than wrapped around an existing native (DX, OpenGL, OpenCL or other) object.

Return Value	*true* when the object is created using AMF, *false* when the object is a native object

AMFData::SetPts

```
void AMF_STD_CALL SetPts(amf_pts pts);
```

Set a timestamp on a memory object. This is applicable to memory objects containing media samples, such as video frames or audio buffers.

Parameter	Description
pts [in]	Timestamp in hundreds of nanoseconds

AMFData::GetPts

```
amf_pts AMF_STD_CALL GetPts();
```

Get a timestamp associated with a memory object. This is applicable to memory objects containing media samples, such as video frames or audio buffers.

Return Value	Timestamp in hundreds of nanoseconds

AMFData::SetDuration

```
void AMF_STD_CALL SetDuration(amf_pts duration);
```

Set duration on a memory object containing a media sample, such as a video frame or an audio buffer.

Parameter	Description
duration [in]	Duration in hundreds of nanoseconds

AMFData::GetDuration

```
amf_pts AMF_STD_CALL GetDuration();
```

Get duration of a media sample, such as a video frame or an audio buffer, stored in the memory object.

Return Value	Duration in hundreds of nanoseconds

2.3.2 Buffers

2.3.2.1 AMFBuffer

The *AMFBuffer* interface provides access to an unordered buffer. Buffers can be located in either host (CPU) or GPU memory.

The *AMFBuffer* interface inherits from *AMFData*.

Include *public/include/core/Buffer.h*

AMFBuffer::SetSize

```
AMF_RESULT AMF_STD_CALL SetSize(amf_size newSize);
```

Change the size of the buffer.

Changing the size of the buffer does not cause memory reallocation. Setting the size to a value larger than the allocated size would cause *SetSize* to fail.

Parameter	**Description**
newSize [in]	Size of the buffer in bytes
Return Value	*AMF_OK* on success *AMF_INVALID_ARG* when the new size exceeds the allocated size of the buffer

AMFBuffer::GetSize

```
amf_size AMF_STD_CALL GetSize();
```

Get buffer size. This method returns either the allocated size or the last size successfully set using *AMFBuffer::SetSize*.

Return Value	Buffer size in bytes

AMFBuffer::GetNative

```
void* AMF_STD_CALL GetNative();
```

Get a pointer to the *AMFBuffer* object's data in host memory, mapping it to host memory when necessary.

Return Value	Pointer to data in host memory

AMFBuffer::AddObserver

```
void AMF_STD_CALL AddObserver(AMFBufferObserver* pObserver);
```

Register the observer interface to be notified when the buffer can be used again.

Parameter	**Description**
pObserver [in]	A pointer to the *AMFBufferObserver* interface to receive notifications

AMFBuffer::RemoveObserver

```
void AMF_STD_CALL RemoveObserver(AMFBufferObserver* pObserver);
```

Unregister an observer previously registered with *AMFBuffer::AddObserver*.

Parameter	**Description**
pObserver [in]	A pointer to the *AMFBufferObserver* interface to stop receiving notifications

2.3.2.2 AMFBufferObserver

The *AMFBufferObserver* interface is used to notify other components that a buffer becomes free from exclusive usage by the component it was submitted to.

When a buffer is submitted to an AMF component as an input resource, the component might require an exclusive access to it for a period of time. When a component releases the buffer, all registered observers receive a notification through the *AMFBufferObserver::OnBufferDataRelease* method, indicating the buffer can be used again. This mechanism is useful when implementing buffer pools that recycle buffers allocated externally.

The *AMFBufferObserver* interface must be implemented by objects observing the buffer.

AMFBufferObserver::OnBufferDataRelease

```
void AMF_STD_CALL OnBufferDataRelease(AMFBuffer* pBuffer);
```

This method is to be implemented by the observer object. It will be called when the buffer becomes free of exclusive access by another component.

Parameter	**Description**
pBuffer [in]	A pointer to the *AMFBuffer* interface

2.3.2.3 AMFAudioBuffer

The *AMFAudioBuffer* interface provides access to a buffer containing audio samples. Buffers can be located in either host (CPU) or GPU memory.

The *AMFAudioBuffer* interface inherits from *AMFData*.

Include *public/include/core/AudioBuffer.h*

AMFBuffer::GetSize

```
amf_size AMF_STD_CALL GetSize();
```

Get buffer size. This method returns either the allocated size or the last size successfully set using *AMFBuffer::SetSize*.

Return Value	Buffer size in bytes

AMFBuffer::GetNative

```
void* AMF_STD_CALL GetNative();
```

Get a pointer to the *AMFBuffer* object's data in host memory, mapping it to host memory when necessary.

Return Value	Pointer to data in host memory

AMFBuffer::GetSampleCount

```
amf_int32 AMF_STD_CALL GetSampleCount();
```

Get the number of audio samples in a buffer.

Return Value	Total number of samples in the buffer

AMFBuffer::GetSampleRate

```
amf_int32 AMF_STD_CALL GetSampleRate();
```

Get the sampling rate of an audio buffer.

Return Value	Sampling rate in samples per second

AMFBuffer::GetSampleSize

```
amf_int32 AMF_STD_CALL GetSampleSize();
```

Get the sample size of an audio buffer.

Return Value	Sample size in bytes

AMFBuffer::GetChannelCount

```
amf_int32 AMF_STD_CALL GetChannelCount();
```

Get the number of audio channels stored in a buffer.

Return Value	Number of audio channels

AMFBuffer::GetChannelLayout

```
amf_uint32 AMF_STD_CALL GetChannelLayout();
```

Get the speaker layout associated with the audio buffer.

Return Value	An ffpmpeg AV_CH_LAYOUT enumeration value

AMFBuffer::AddObserver

```
void AMF_STD_CALL AddObserver(AMFAudioBufferObserver* pObserver);
```

Register the observer interface to be notified when the buffer can be used again.

Parameter	**Description**
pObserver [in]	A pointer to the *AMFAudioBufferObserver* interface to receive notifications

AMFBuffer::RemoveObserver

```
void AMF_STD_CALL RemoveObserver(AMFAudioBufferObserver* pObserver);
```

Unregister an observer previously registered with *AMFAudioBuffer::AddObserver*.

Parameter	**Description**
pObserver [in]	A pointer to the *AMFAudioBufferObserver* interface to stop receiving notifications

2.3.2.4 AMFAudioBufferObserver

The *AMFAudioBufferObserver* interface is used to notify other components that a buffer becomes free from exclusive usage by the component it was submitted to.

When a buffer is submitted to an AMF component as an input resource, the component might require an exclusive access to it for a period of time. When a component releases the buffer, all registered observers receive a notification through the *AMFAudioBufferObserver::OnBufferDataRelease* method, indicating the buffer can be used again. This mechanism is useful when implementing buffer pools that recycle buffers allocated externally.

The *AMFAudioBufferObserver* interface must be implemented by objects observing the buffer.

AMFBufferObserver::OnBufferDataRelease

```
void AMF_STD_CALL OnBufferDataRelease(AMFBuffer* pBuffer);
```

This method is to be implemented by the observer object. It will be called when the buffer becomes free of exclusive access by another component.

Parameter	**Description**
pBuffer [in]	A pointer to the *AMFBuffer* interface

2.3.3 Surfaces

2.3.3.1 AMFSurface

The *AMFSurface* interface abstracts a memory buffer containing a 2D image (typically a video frame) accessible by the GPU. The structure of the buffer depends on the surface type and format. Memory buffers associated with a surface may be stored in either GPU or host memory and consist of one or more planes accessible through the *AMFPlane* interface.

AMFSurface inherits from *AMFData*. *AMFSurface* objects are generally not thread-safe.

Include *public/include/core/Surface.h*

Surface Formats

Surface format defines how pixel data is stored in memory. Surface format is described by the *AMF_SURFACE_FORMAT* enumeration.

The following formats are supported in AMF:

Format	Description
AMF_SURFACE_UNKNOWN	Format unknown/undefined
AMF_SURFACE_NV12	Y plane of width * height size, packed UV plane of width/2 * height/2 size, 8 bits per component
AMF_SURFACE_YV12	Y plane of width * height size, V plane of width/2 * height/2 size, U plane of width/2 * height/2, 8 bits per component

AMF_SURFACE_BGRA	packed - 8 bits per component
AMF_SURFACE_ARGB	packed - 8 bits per component
AMF_SURFACE_RGBA	packed - 8 bits per component
AMF_SURFACE_GRAY8	single component - 8 bits
AMF_SURFACE_YUV420P	Y plane of width * height size, U plane of width/2 * height/2 size, V plane of width/2 * height/2, 8 bits per component
AMF_SURFACE_U8V8	double component - 8 bits per component
AMF_SURFACE_YUY2	YUY2: Byte 0=8-bit Y'0; Byte 1=8-bit Cb; Byte 2=8-bit Y'1; Byte 3=8-bit Cr
AMF_SURFACE_P010	Y plane of width * height, packed UV plane of width/2 * height/2, 10 bits per component (16 allocated, upper 10 bits are used)
AMF_SURFACE_RGBA_F16	packed - 16 bit float per component

Frame Types

The type of a video frame contained in a surface is defined using the *AMF_FRAME_TYPE* enumeration. The values of this enumeration can be used to describe a specific frame as well as the entire video sequence. The *AMF_FRAME_TYPE* enumeration is defined as follows:

Value	Description
AMF_FRAME_STEREO_FLAG	The surface contains a part of a stereoscopic frame
AMF_FRAME_LEFT_FLAG	The surface contains the left eye portion of a stereoscopic frame, includes *AMF_FRAME_STEREO_FLAG*
AMF_FRAME_RIGHT_FLAG	The surface contains the right eye portion of a stereoscopic frame, includes *AMF_FRAME_STEREO_FLAG*
AMF_FRAME_BOTH_FLAG	The surface contains the entire stereoscopic frame (both eyes) , includes *AMF_FRAME_STEREO_FLAG*
AMF_FRAME_INTERLEAVED_FLAG	The surface contains an interlaced image
AMF_FRAME_FIELD_FLAG	The surface contains a single field of an interlaced image
AMF_FRAME_EVEN_FLAG	The surface contains the even field of an interlaced image
AMF_FRAME_ODD_FLAG	The surface contains the odd field of an interlaced image
AMF_FRAME_UNKNOWN	Frame format is unknown
AMF_FRAME_PROGRESSIVE	The frame is progressive
AMF_FRAME_INTERLEAVED_EVEN_FIRST	The sequence is interlaced with the even field preceding the odd field
AMF_FRAME_INTERLEAVED_ODD_FIRST	The sequence is interlaced with the odd field preceding the even field
AMF_FRAME_FIELD_SINGLE_EVEN	The surface contains a single even field
AMF_FRAME_FIELD_SINGLE_ODD	The surface contains a single odd field
AMF_FRAME_STEREO_LEFT	Same as *AMF_FRAME_LEFT_FLAG*
AMF_FRAME_STEREO_RIGHT	Same as *AMF_FRAME_RIGHT_FLAG*
AMF_FRAME_STEREO_BOTH	Same as *AMF_FRAME_BOTH_FLAG*
AMF_FRAME_INTERLEAVED_EVEN_FIRST_STEREO_LEFT	A combination of *AMF_FRAME_INTERLEAVED_EVEN_FIRST and*

	AMF_FRAME_LEFT_FLAG
AMF_FRAME_INTERLEAVED_EVEN_FIRST_STEREO_RIGHT	A combination of *AMF_FRAME_INTERLEAVED_EVEN_FIRST and AMF_FRAME_RIGHT_FLAG*
AMF_FRAME_INTERLEAVED_EVEN_FIRST_STEREO_BOTH	A combination of *AMF_FRAME_INTERLEAVED_EVEN_FIRST and AMF_FRAME_BOTH_FLAG*
AMF_FRAME_INTERLEAVED_ODD_FIRST_STEREO_LEFT	A combination of *AMF_FRAME_INTERLEAVED_ODD_FIRST and AMF_FRAME_LEFT_FLAG*
AMF_FRAME_INTERLEAVED_ODD_FIRST_STEREO_RIGHT	A combination of *AMF_FRAME_INTERLEAVED_ODD_FIRST and AMF_FRAME_RIGHT_FLAG*
AMF_FRAME_INTERLEAVED_ODD_FIRST_STEREO_BOTH	A combination of *AMF_FRAME_INTERLEAVED_ODD_FIRST and AMF_FRAME_BOTH_FLAG*

AMFSurface::GetFormat()

```
AMF_SURFACE_FORMAT AMF_STD_CALL GetFormat();
```

Get the format of the surface. Refer to Surface Formats for more information about various surface formats.

Return Value	Surface format.

AMFSurface::GetPlanesCount()

```
amf_size AMF_STD_CALL GetPlanesCount();
```

Get the number of planes in the surface. The number of planes depends on the surface format. Refer to Surface Formats for more information on the surface structure for different formats.

Return Value	The number of planes in the surface

AMFSurface::GetPlaneAt()

```
AMFPlane* AMF_STD_CALL GetPlaneAt(amf_size index);
```

Obtain a pointer to the specific plane by index.

This method does not increment the reference count on the *AMFPlane* interface returned.

Parameter	**Description**
index [in]	A 0-based index of the requested plane
Return Value	A pointer to the *AMFPlane* interface

AMFSurface::GetPlane()

```
AMFPlane* GetPlane(AMF_PLANE_TYPE type);
```

Obtain a pointer to the specific plane by plane type.

This method does not increment the reference count on the *AMFPlane* interface returned.

Parameter	Description
type [in]	The type of the requested plane
Return Value	A pointer to the *AMFPlane* interface

AMFSurface::GetFrameType()

```
AMF_FRAME_TYPE AMF_STD_CALL GetFrameType();
```

Get the type of a frame stored in the surface. Refer to the Frame Types section for more information on frame types.

Return Value	The type of the frame stored in the surface

AMFSurface::SetFrameType()

```
void AMF_STD_CALL SetFrameType(AMF_FRAME_TYPE type);
```

Set the type of the frame stored in the surface. Refer to the Frame Types section for more information on frame types.

Parameter	Description
type [in]	The type of the frame stored in the surface

AMFSurface::SetCrop()

```
AMF_RESULT AMF_STD_CALL SetCrop(amf_int32 x, amf_int32 y,
       amf_int32 width, amf_int32 height);
```

Set the cropping region on the surface. Pixels outside of the cropping region will be ignored by all manipulations on the surface.

Parameter	Description
x [in]	Horizontal offset
y [in]	Vertical offset
width [in]	Crop width
height [in]	Crop height
Return Value	*AMF_OK* on success, *AMF_INVALID_ARG* otherwise

AMFSurface::AddObserver

```
void AMF_STD_CALL AddObserver(AMFSurfaceObserver* pObserver);
```

Register the observer interface to be notified when the buffer can be used again.

Parameter	Description
pObserver [in]	A pointer to the *AMFSurfaceObserver* interface to receive notifications

AMFSurface::RemoveObserver

```
void AMF_STD_CALL RemoveObserver(AMFSurfaceObserver* pObserver);
```

Unregister an observer previously registered with *AMFSurface::AddObserver*.

Parameter	Description
pObserver [in]	A pointer to the *AMFSurfaceObserver* interface to stop receiving notifications

2.3.3.2 AMFSurfaceObserver

The *AMFSurfaceObserver* interface is used to notify other components that a surface becomes free from exclusive usage by the component it was submitted to.

When a surface is submitted to an AMF component as an input resource, the component might require an exclusive access to it for a period of time. When a component releases the surface, all registered observers receive a notification through the *AMFSurfaceObserver::OnSurfaceDataRelease* method, indicating the surface can be used again. This mechanism is useful when implementing surface pools that recycle externally allocated surfaces.

The *AMFSurfaceObserver* interface must be implemented by objects observing the surface.

AMFSurfaceObserver::OnSurfaceDataRelease

```
void AMF_STD_CALL OnSurfaceDataRelease(AMFSurface* pSurface);
```

This method is to be implemented by the observer object. It will be called when the surface becomes free of exclusive access by another component.

Parameter	Description
pSurface [in]	A pointer to the *AMFSurface* interface

2.3.3.3 AMFPlane

The *AMFPlane* interface provides access to a single plane of a surface. A pointer to the *AMFPlane* interface can be obtained using the *GetPlane* and *GetPlaneAt* methods of the *AMFSurface* interface. Any *AMFSurface* object contains at least one plane. The number of planes in a surface is determined by Surface Format.

Plane Types

Plane types are defined using the *AMF_PLANE_TYPE* enumeration and can be one of the following:

Value	Description
AMF_PLANE_PACKED	All single-plane packed formats, such as BGRA, YUY2, etc.
AMF_PLANE_Y	The Y plane for all multi-plane formats
AMF_PLANE_UV	The UV plane for formats with combined UV planes
AMF_PLANE_U	The U plane
AMF_PLANE_V	The V plane

AMFPlane::GetType()

```
AMF_PLANE_TYPE AMF_STD_CALL GetType();
```

Get plane type.

Return Value	Current plane type

AMFPlane::GetNative()

```
void* AMF_STD_CALL GetNative();
```

Obtain a native interface to the underlying memory.

The return value of this method depends on how the surface containing the plane was created. For DirectX objects, such as DirectX 9 surfaces and DirectX 11 textures, this method returns a pointer to the underlying native interface (*IDirect3DSurface9* or *ID3D11Texture2D*). For OpenCL and Compute objects *GetNative* returns a handle to the memory buffer containing the plane.

For multi-plane surfaces the value returned may or may not be the same for all planes belonging to the same surface, therefore no assumptions should be made about the value returned by *GetNative*. Native objects should be explicitly requested for every plane and cast to the appropriate type. For DirectX objects, when a pointer to a COM interface is returned, *GetNative* does not call *IUnknown::AddRef* on the interface being returned.

Return Value	A native accessor to the plane

AMFPlane::GetPixelSizeInBytes()

```
amf_int32 AMF_STD_CALL GetPixelSizeInBytes();
```

Get the size of a pixel in the plane in bytes. The size of a pixel in each plane of a surface depends on the format of the surface.

Return Value	The size of a pixel in the plane in bytes

AMFPlane::GetOffsetX()

```
amf_int32 AMF_STD_CALL GetOffsetX();
```

Get the horizontal offset of the crop region in the plane.

The crop region can be set on a surface using *AMFSurface::SetCrop* method. The crop region would be applied to all planes of the surface according to the format of the surface. Crop regions cannot be applied to individual planes independently.

Return Value	The horizontal offset of the crop region in pixels

AMFPlane::GetOffsetY()

```
amf_int32 AMF_STD_CALL GetOffsetY();
```

Get the vertical offset of the crop region in the plane.

The crop region can be set on a surface using *AMFSurface::SetCrop* method. The crop region would be applied to all planes of the surface according to the format of the surface. Crop regions cannot be applied to individual planes independently.

Return Value	The vertical offset of the crop region in pixels

AMFPlane::GetWidth()

```
amf_int32 AMF_STD_CALL GetWidth();
```

Get the width of the crop region in the plane.

The crop region can be set on a surface using *AMFSurface::SetCrop* method. The crop region would be applied to all planes of the surface according to the format of the surface. Crop regions cannot be applied to individual planes independently.

When the crop region is not set, the value returned by *GetWidth* is based on the full width of the surface containing the plane and its pixel format.

Return Value	The width of the crop region in pixels

AMFPlane::GetHeight()

```
amf_int32 AMF_STD_CALL GetHeight();
```

Get the height of the crop region in the plane.

The crop region can be set on a surface using *AMFSurface::SetCrop* method. The crop region would be applied to all planes of the surface according to the format of the surface. Crop regions cannot be applied to individual planes independently.

When the crop region is not set, the value returned by *GetHeight* is based on the full height of the surface containing the plane and its pixel format.

Return Value	The height of the crop region in pixels

AMFPlane::GetHPitch()

```
amf_int32 AMF_STD_CALL GetHPitch();
```

Get the horizontal pitch of the plane. Horizontal pitch is the amount of memory a single scan line, including any padding, occupies.

Return Value	Horizontal pitch of the plane in bytes

AMFPlane::GetVPitch()

```
amf_int32 AMF_STD_CALL GetVPitch();
```

Get the vertical pitch of the plane. Vertical pitch is the number of scan lines, including any padding, a plane occupies. Vertical pitch is always a multiple of horizontal pitch.

Return Value	Vertical pitch of the plane in scan lines

AMFPlane::IsTiled

```
bool AMF_STD_CALL IsTiled();
```

Determine whether the physical memory storing the plane is contiguous or tiled.

Return Value	*true* when the memory is tiled, *false* when the memory is contiguous

2.4 Device Abstraction

2.4.1 AMFContext

The *AMFContext* interface serves as an entry point to most AMF functionality, acting as a facility to create and initialize device-specific resources. It also abstracts the underlying platform-specific technologies, providing a consistent API across DirectX9, DirectX11, OpenGL, OpenCL, XV, Android.

Context Initialization

```
AMF_RESULT AMF_STD_CALL InitDX9(void* pDX9Device);
AMF_RESULT AMF_STD_CALL InitDX11(void* pDX11Device, AMF_DX_VERSION dxVersionRequired);
AMF_RESULT AMF_STD_CALL InitOpenCL(void* pCommandQueue);
AMF_RESULT AMF_STD_CALL InitOpenGL(amf_handle hOpenGLContext, amf_handle hWindow,
       amf_handle hDC);
```

The *Init* methods initialize the *AMFContext* object to use a specific technology, such as DirectX, OpenGL, OpenCL, XV, etc. A single context can be initialized once for a particular technology, but it can be initialized to use different technologies at the same time – you can initialize the same context to use DirectX9 and OpenGL at the same time, for example.

The parameters passed to *Init* methods depend on the underlying technology. Usually they include a device or device context handle or a pointer to an interface providing access to the device, as well as other values, such as the version of DirectX required, a window handle, etc.

Parameter	**Description**
pDX9Device [in]	A pointer to the *IDirectX9Device* interface
pDX11Device [in]	A pointer to the *ID3D11Device* interface
pCommandQueue [in]	An OpenCL command queue handle of *cl_command_queue* type, returned by the *clCreateCommandQueue* function
hOpenGLContext [in]	An OpenGL context handle returned by the *wglCreateContext* function
dxVersionRequired [in]	The minimum DirectX version requested (defaults to DX11.0)
hWindow [in]	A Win32 handle (HWND) of the output window. When set to NULL, the desktop window will be used.
hDC [in]	A Win32 device context (HDC). When set to NULL, the default device context will be obtained from the window passed through the *hWindow* parameter
Return Value	*AMF_OK* on success *AMF_DIRECTX_FAILED*, *AMF_OPENCL_FAILED*, *AMF_GLX_FAILED* on failure *AMF_ALREADY_INITIALIZED* when the context re-initialization for the same technology was attempted

Get Native Device Interfaces

```
void* AMF_STD_CALL GetDX9Device(AMF_DX_VERSION dxVersionRequired);
void* AMF_STD_CALL GetDX11Device(AMF_DX_VERSION dxVersionRequired);
void* AMF_STD_CALL GetOpenCLContext();
void* AMF_STD_CALL GetOpenCLCommandQueue();
void* AMF_STD_CALL GetOpenCLDeviceID();
```

Obtain a native device interface.

These methods return the native device used to initialize the context. Their return values need to be explicitly cast to the native interface or handle.

Note that methods returning a pointer to a COM interface, such as *GetDX9Device* and *GetDX11Device*, do not increment the reference counter on the interface they return.

Return Value	Native device interface or handle

AMFContext::Terminate()

```
AMF_RESULT AMF_STD_CALL Terminate();
```

Terminate the context. The context can be initialized again after it has been terminated.

Return Value	*AMF_OK* on success

Device Lock

```
AMF_RESULT AMFContext::LockDX9();
AMF_RESULT AMFContext::LockDX11();
AMF_RESULT AMFContext::LockOpenCL();
AMF_RESULT AMFContext::LockOpenGL();
```

Lock the device associated with the context for exclusive use.

Return Value	*AMF_OK* on success *AMF_NOT_INITIALIZED* when called on a context which hasn't been initialized for the specific platform

Device Unlock

```
AMF_RESULT AMFContext::UnlockDX9();
AMF_RESULT AMFContext::UnlockDX11();
AMF_RESULT AMFContext::UnlockOpenCL();
AMF_RESULT AMFContext::UnlockOpenGL();
```

Unlock the device associated with the context that was previously locked with a corresponding *Lock* method.

Return Value	*AMF_OK* on success *AMF_NOT_INITIALIZED* when called on a context which hasn't been initialized for the specific platform

AMFContext::AllocBuffer()

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

Allocate a buffer object. The context must be initialized for the technology matching the memory type specified by the *type* parameter.

Parameter	**Description**
type [in]	Memory type
size [in]	Buffer size in bytes
ppBuffer [out]	A pointer to the location to receive a pointer to the *AMFBuffer* interface
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFContext::CreateBufferFromHostNative()

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

Wrap an existing buffer in host (CPU) memory in an *AMFBuffer* object.

Parameter	Description
pHostBuffer [in]	A pointer to the buffer in host memory
size [in]	Buffer size in bytes
ppBuffer [out]	A pointer to the location to receive a pointer to the *AMFBuffer* interface
pObserver [in]	A pointer to an object implementing the *AMFBufferObserver* interface to receive a notification when the corresponding *AMFBuffer* object is being destroyed
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFContext::CreateBufferFromOpenCLNative()

```
AMF_RESULT AMF_STD_CALL CreateBufferFromOpenCLNative(void* pCLBuffer, amf_size size,
        AMFBuffer** ppBuffer);
```

Wrap an existing OpenCL buffer in an *AMFBuffer* object.

Parameter	Description
pHostBuffer [in]	A pointer to the buffer in host memory
size [in]	Buffer size in bytes
ppBuffer [out]	A pointer to the location to receive a pointer to the *AMFBuffer* interface
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFContext::AllocSurface()

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

Allocate a surface object. The context must be initialized for the technology matching the memory type specified by the *type* parameter.

Parameter	Description
type [in]	Memory type
format [in]	Pixel format
width [in]	Surface width in pixels
height [in]	Surface height in scan lines
ppSurface [out]	A pointer to the location to receive a pointer to the *AMFSurface* interface
	AMF_OK on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type, format or surface sizes are invalid

AMFContext::CreateSurfaceFromDX9Native

```
AMF_RESULT AMF_STD_CALL CreateSurfaceFromDX9Native(void* pDX9Surface,
        AMFSurface** ppSurface, AMFSurfaceObserver* pObserver);
```

Wrap an existing native DirectX9 2D surface object in an *AMFSurface* object. The context must be initialized for DirectX9.

Parameter	Description
pDX9Surface [in]	A pointer to the IDirect3DSurface9 interface

ppSurface [out]	A pointer to the location to receive a pointer to the *AMFSurface* interface
pObserver [in]	A pointer to an object implementing the *AMFSurfaceObserver* interface to receive a notification when the corresponding *AMFSurface* object is being destroyed
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFContext::CreateSurfaceFromDX11Native

```
AMF_RESULT AMF_STD_CALL CreateSurfaceFromDX11Native(void* pDX11Surface,
        AMFSurface** ppSurface, AMFSurfaceObserver* pObserver);
```

Wrap an existing native DirectX11 2D texture object in an *AMFSurface* object. The context must be initialized for DirectX11.

Parameter	**Description**
pDX11Surface [in]	A pointer to the ID3D11Texture2D interface
ppSurface [out]	A pointer to the location to receive a pointer to the *AMFSurface* interface
pObserver [in]	A pointer to an object implementing the *AMFSurfaceObserver* interface to receive a notification when the corresponding *AMFSurface* object is being destroyed
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFContext::CreateSurfaceFromHostNative

```
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);
```

Wrap an existing buffer in host memory in an *AMFSurface* object.

Parameter	**Description**
format [in]	Pixel format
width [in]	Width in pixels
height [in]	Height in scan lines
hPitch [in]	Horizontal pitch in bytes
vPitch [in]	Vertical pitch in scan lines
pData [in]	A pointer to the buffer in host memory
ppSurface [out]	A pointer to the location to receive a pointer to the *AMFSurface* interface
pObserver [in]	A pointer to an object implementing the *AMFSurfaceObserver* interface to receive a notification when the corresponding *AMFSurface* object is being destroyed
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFContext::CreateSurfaceFromOpenGLNative

```
AMF_RESULT AMF_STD_CALL CreateSurfaceFromOpenGLNative(AMF_SURFACE_FORMAT format,
       amf_handle hGLTextureID, AMFSurface** ppSurface, AMFSurfaceObserver* pObserver);
```

Wrap an existing native OpenGL texture in an *AMFSurface* object. The context must be initialized for OpenGL.

Parameter	**Description**
format [in]	Pixel format
hGLTextureID [in]	OpenGL texture ID
ppSurface [out]	A pointer to the location to receive a pointer to the *AMFSurface* interface
pObserver [in]	A pointer to an object implementing the *AMFSurfaceObserver* interface to receive a notification when the corresponding *AMFSurface* object is being destroyed
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFContext::CreateSurfaceFromOpenCLNative

```
AMF_RESULT AMF_STD_CALL CreateSurfaceFromOpenCLNative(AMF_SURFACE_FORMAT format, amf_int32
       width, amf_int32 height, void** pClPlanes, AMFSurface** ppSurface,
       AMFSurfaceObserver* pObserver);
```

Wrap an existing native OpenCL surface in an *AMFSurface* object. The context must be initialized for OpenCL.

Parameter	**Description**
format [in]	Pixel format
width [in]	Width in pixels
height [in]	Height in scan lines
pCLPlanes [in]	A pointer to an array of OpenCL handles to buffers representing planes. The number of planes is defined by the pixel format
ppSurface [out]	A pointer to the location to receive a pointer to the *AMFSurface* interface
pObserver [in]	A pointer to an object implementing the *AMFSurfaceObserver* interface to receive a notification when the corresponding *AMFSurface* object is being destroyed
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFContext::GetOpenCLComputeFactory

```
AMF_RESULT AMF_STD_CALL GetOpenCLComputeFactory(AMFComputeFactory** ppFactory);
```

Obtain a pointer to the AMF Compute class factory. The context must be initialized for OpenCL.

Parameter	**Description**
ppFactory [out]	A pointer to the location to receive a pointer to the *AMFComputeFactory* interface
Return Value	*AMF_OK* on success *AMF_INVALID_ARG* when *ppFactory* is *nullptr*

AMFContext::GetCompute

```
AMF_RESULT AMF_STD_CALL GetCompute(AMF_MEMORY_TYPE memType, AMFCompute** ppCompute);
```

Create an AMF Compute object for a specific memory type.

The *AMFContext* object must be initialized with a call to one of the *Init** methods for OpenCL, DirectX9 or DirectX11. If it has not been initialized for the technology specified by the *memType* parameter, an implicit initialization will be performed.

Parameter	**Description**
memType [in]	Memory type. Can be one of the following: *AMF_MEMORY_OPENCL* – for OpenCL *AMF_MEMORY_COMPUTE_FOR_DX9* – for AMF Compute on DirectX9 *AMF_MEMORY_COMPUTE_FOR_DX11* – for AMF Compute on DirectX11
ppFactory [out]	A pointer to the location to receive a pointer to the AMFComputeFactory interface
Return Value	*AMF_OK* on success *AMF_INVALID_ARG* when *ppFactory* is *nullptr* or when *memType* is set to a value different from *AMF_MEMORY_OPENCL*, *AMF_MEMORY_COMPUTE_FOR_DX9 or AMF_MEMORY_COMPUTE_FOR_DX11*

2.5 AMF Compute

The AMF Compute API provides an abstraction layer for running OpenCL kernels.

In addition to the standard OpenCL implementation AMD devices allow to execute OpenCL kernels in different way, which provides a more efficient interop with DirectX. AMF Compute provides a uniform API to utilize both subsystems. Note that interop with DirectX11 is available only on Windows 8.1 and newer versions of Microsoft Windows.

Applications should choose between standard OpenCL and AMF Compute technologies based on performance requirements. AMF Compute is recommended when interop with DirectX is heavily utilized.

Most of the AMF Compute functions can be accessed through the AMFCompute interface. A pointer to the AMFCompute interface can be obtained by calling AMFContext*::GetCompute* or the AMFDeviceCompute*::CreateCompute* or AMFDeviceCompute*::CreateComputeEx* methods.

Compute devices can be enumerated using the AMFComputeFactory interface. To obtain a pointer to the *AMFComputeFactory* interface, call AMFContext*::GetOpenCLComputeFactory*. To utilize the standard OpenCL API through AMF Compute, initialize the context with a call to *AMFContext::InitOpenCL* or *AMFContext::InitOpenCLEx.* When the context is initialized with *AMFContext::InitDX9* or *AMFContext::InitDX11*, AMF Compute will be used.

OpenCL kernels can be submitted to and executed in an AMF Compute queue. Kernels need to be registered with AMF before they can be used. Kernel registration is performed using the AMFPrograms interface, a pointer to which can be obtained by calling AMFFactory*::GetPrograms*.

2.5.1 AMFComputeFactory

The *AMFComputeFactory* interface allows to enumerate Compute devices available in the system. The *AMFComputeFactory* interface inherits from AMFInterface.

Include *public/include/core/ComputeFactory.h*

AMFComputeFactory::GetDeviceCount

```
amf_int32 AMF_STD_CALL GetDeviceCount();
```

Get the total count of Compute devices available in the system.

Return Value	The total number of AMF Compute devices available

AMFComputeFactory::GetDeviceAt

```
AMF_RESULT AMF_STD_CALL GetDeviceAt(amf_int32 index, AMFDeviceCompute **ppDevice);
```

Obtain a pointer to a specific AMF Compute device.

Parameter	**Description**
index [in]	Device zero-based index
ppDevice [out]	A pointer to the location to receive a pointer to the *AMFDeviceCompute* interface
Return Value	*AMF_OK* on success *AMF_INVALID_ARG* when *ppDevice* is *nullptr*

2.5.2 AMFDeviceCompute

The *AMFDeviceCompute* interface provides access to the functionality of an AMF Compute device object.

The *AMFDeviceCompute* interface inherits from *AMFPropertyStorage*.

Include *public/include/core/ComputeFactory.h*

AMFDeviceCompute::GetNativePlatform

```
void* AMF_STD_CALL GetNativePlatform();
```

Get the native AMF Compute platform descriptor. The return value of this method should be treated as an opaque handle.

Return Value	Native AMF Compute platform descriptor.

AMFDeviceCompute::GetNativeDeviceID

```
void* AMF_STD_CALL GetNativeDeviceID();
```

Get the native AMF Compute device ID. The return value of this method should be treated as an opaque handle.

Return Value	Native AMF Compute device ID.

AMFDeviceCompute::GetNativeContext

```
void* AMF_STD_CALL GetNativeContext();
```

Get the native AMF Compute context. The return value of this method should be treated as an opaque handle.

Return Value	Native AMF Compute context.

AMFDeviceCompute::CreateCompute AMFDeviceCompute::CreateComputeEx

```
AMF_RESULT AMF_STD_CALL CreateCompute(void* reserved, AMFCompute** ppCompute);
AMF_RESULT AMF_STD_CALL CreateComputeEx(void* pCommandQueue, AMFCompute** ppCompute);
```

Create an AMF Compute object and obtain the AMFCompute interface pointer.

CreateCompute uses the default command queue created by AMFContext*::InitOpenCL()*. *CreateComputeEx* allows to specify a command queue created externally.

Parameter	**Description**
reserved [in]	Reserved. Must be set to *nullptr*
pCommandQueue [in]	A handle to an OpenCL command queue
ppCompute [out]	A pointer to a location to receive a pointer to the AMFCompute interface
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when *ppCompute* is *nullptr*

2.5.3 AMFPrograms

The *AMFPrograms* interface is used to compile and register AMF Compute kernels with AMF. AMF Compute kernels use the same syntax as OpenCL kernels.

A pointer to the *AMFPrograms* interface can be obtained by calling the AMFFactory*::GetPrograms* method. The *AMFPrograms* interface is not reference-counted and represents a global object, which maintains a registry of all AMF Compute kernels used by the application. Do not call *delete* on the *AMFPrograms* interface.

Include *public/include/core/Compute.h*

AMFPrograms::RegisterKernelSourceFile

```
AMF_RESULT AMF_STD_CALL RegisterKernelSourceFile(AMF_KERNEL_ID* pKernelID,
       const wchar_t* trace_name, const char* kernelName, const wchar_t* filepath, const
       char* options);
```

Compile an AMF Compute kernel from a source file and register it with AMF Compute.

Parameter	**Description**
pKernelID [out]	A pointer to the location to receive a unique kernel ID assigned by AMF
trace_name [in]	A unique human-readable string to uniquely identify a specific kernel. Used in performance trace enabled by AMFDebug*::EnablePerformanceMonitor*
kernelName [in]	Kernel name in the source file
filepath [in]	Source file path
options [in]	Kernel options passed to *clBuildProgram*
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFPrograms::RegisterKernelSource

```
AMF_RESULT AMF_STD_CALL RegisterKernelSource(AMF_KERNEL_ID* pKernelID,
       const wchar_t* kernelid_name, const char* kernelName, amf_size dataSize,
       const amf_uint8* data, const char* options);
```

Compile an AMF Compute kernel from source located in a memory buffer and register it with AMF Compute.

Parameter	Description
pKernelID [out]	A pointer to the location to receive a unique kernel ID assigned by AMF
trace_name [in]	A unique human-readable string to uniquely identify a specific kernel. Used in performance trace enabled by *AMFDebug::EnablePerformanceMonitor*
kernelName [in]	Kernel name in the source file
dataSize [in]	The size of the buffer containing the kernel's source code in bytes
data [in]	A pointer to the buffer containing the kernel's source code
options [in]	Kernel options passed to *clBuildProgram*
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFPrograms::RegisterKernelBinary

```
AMF_RESULT AMF_STD_CALL RegisterKernelBinary(AMF_KERNEL_ID* pKernelID,
       const wchar_t* kernelid_name, const char* kernelName, amf_size dataSize,
       const amf_uint8* data, const char* options);
```

Load and register a precompiled kernel located in a memory buffer.

Parameter	Description
pKernelID [out]	A pointer to the location to receive a unique kernel ID assigned by AMF
trace_name [in]	A unique human-readable string to uniquely identify a specific kernel. Used in performance trace enabled by *AMFDebug::EnablePerformanceMonitor*
kernelName [in]	Kernel name in the source file
dataSize [in]	The size of the buffer containing the kernel's compiled code in bytes
data [in]	A pointer to the buffer containing the kernel's compiled code
options [in]	Kernel options passed to *clBuildProgram*
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

2.5.4 AMFCompute

The *AMFCompute* interface provides access to the functionality of an OpenCL command queue.

The *AMFCompute* interface inherits from *AMFInterface*.

Include *public/include/core/Compute.h*

AMFCompute::GetMemoryType

```
AMF_MEMORY_TYPE AMF_STD_CALL GetMemoryType();
```

Get the type of memory associated with the *AMFCompute* object.

Memory type returned depends on how the *AMFCompute* object was initialized. For objects created with *AMFContext::GetCompute*, the value passed to *GetCompute* will be returned.

Return Value	Memory type. Can be one of the following values: *AMF_MEMORY_OPENCL* – for OpenCL *AMF_MEMORY_COMPUTE_FOR_DX9* – for AMF Compute on DirectX9 *AMF_MEMORY_COMPUTE_FOR_DX11* – for AMF Compute on DirectX11

AMFCompute::GetNativeContext

```
void* AMF_STD_CALL GetNativeContext();
```

Return a handle to the native context associated with the AMF Compute command queue.

Return Value	Handle to the native context

AMFCompute::GetNativeDeviceID

```
void* AMF_STD_CALL GetNativeDeviceID();
```

Return a handle to the native device ID associated with the AMF Compute command queue.

Return Value	Handle to the native device ID

AMFCompute::GetNativeCommandQueue

```
void* AMF_STD_CALL GetNativeCommandQueue();
```

Return a handle to the native command queue associated with the AMF Compute object.

Return Value	Handle to the native command queue

AMFCompute::GetKernel

```
AMF_RESULT AMF_STD_CALL GetKernel(AMF_KERNEL_ID kernelID, AMFComputeKernel** kernel);
```

Load an AMF Compute kernel from the global registry and associate it with the *AMFCompute* object.

Parameter	**Description**
kernelID [in]	A unique kernel ID returned by one of the *AMFPrograms* methods
kernel [out]	A pointer to a location to receive a pointer to the *AMFComputeKernel* interface
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::PutSyncPoint

```
AMF_RESULT AMF_STD_CALL PutSyncPoint(AMFComputeSyncPoint** ppSyncPoint);
```

Insert a synchronization point into the AMF Compute queue.

A synchronization point allows the CPU to wait for the completion and query the status of a particular operation submitted to an AMF Compute queue.

Return Value	A pointer to a location to receive a pointer to the *AMFComputeSyncPoint*

interface

AMFCompute::FlushQueue

```
AMF_RESULT AMF_STD_CALL FlushQueue();
```

Trigger the AMF Compute queue to immediately start executing submitted tasks.

Under normal conditions the GPU decides when to start executing tasks submitted to the queue. Submitting a task to the queue does not guarantee that the GPU execute it immediately after submission. Flushing the queue triggers immediate execution of all tasks submitted up to the moment of the call.

When GPU profiling is enabled with *AMFDebug::EnablePerformanceMonitor*, flushing the queue also triggers profiling output messages to be dumped.

Return Value	*AMF_OK*

AMFCompute::FinishQueue

```
AMF_RESULT AMF_STD_CALL FinishQueue();
```

Trigger the AMF Compute queue to immediately start executing submitted tasks and wait for their completion.

Return Value	*AMF_OK*

AMFCompute::FillPlane

```
AMF_RESULT AMF_STD_CALL FillPlane(AMFPlane* pPlane, const amf_size origin[3],
       const amf_size region[3], const void* pColor);
```

Fill a surface plane with a solid color.

The *origin* and the *region* parameters represent the 3D coordinates and the size of the area of the plane to be filled. For 2D planes *origin[2]* and *region[2]* must be set to 0.

The fill color is a four component RGBA floating-point color value if the image channel data type is not an unnormalized signed and unsigned integer type, is a four component signed integer value if the image channel data type is an unnormalized signed integer type and is a four component unsigned integer value if the image channel data type is an unormalized unsigned integer type.

Parameter	Description
pPlane [in]	A pointer to an *AMFPlane* object to be filled
origin [in]	A triplet specifying the origin of a rectangular area in the plane to be filled
region [in]	A triplet specifying the size of a rectangular area in the plane to be filled
pColor [in]	Fill color
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::FillBuffer

```
AMF_RESULT AMF_STD_CALL FillBuffer(AMFBuffer* pBuffer, amf_size dstOffset,
        amf_size dstSize, const void* pSourcePattern, amf_size patternSize);
```

Fill a buffer object with a repeating pattern.

When the destination size specified with the *dstSize* parameter is greater than the pattern size specified with the *patternSize* parameter, the pattern will be repeated. When the destination size is not a multiple of pattern size, the last copy of the pattern at the destination will be truncated.

Parameter	Description
pBuffer [in]	A pointer to an *AMFBuffer* object to be filled
dstOffset [in]	Destination offset in bytes
dstSize [in]	Destination size in bytes
pSourcePattern [in]	A pointer to the pattern to fill the buffer with
patternSize [in]	Pattern size in bytes
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::ConvertPlaneToBuffer

```
AMF_RESULT AMF_STD_CALL ConvertPlaneToBuffer(AMFPlane* pSrcPlane,
        AMFBuffer** ppDstBuffer);
```

Create an *AMFBuffer* object and attach a plane to it. Both the buffer and the source plane share the same physical memory. The memory itself is not owned by any of the objects and would get freed only after the last object referencing it is destroyed.

Parameter	Description
pBuffer [in]	A pointer to an *AMFBuffer* object to be filled
dstOffset [in]	Destination offset in bytes
dstSize [in]	Destination size in bytes
pSourcePattern [in]	A pointer to the pattern to fill the buffer with
patternSize [in]	Pattern size in bytes
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::CopyBuffer

```
AMF_RESULT AMF_STD_CALL CopyBuffer(AMFBuffer* pSrcBuffer, amf_size srcOffset,
        amf_size size, AMFBuffer* pDstBuffer, amf_size dstOffset);
```

Copy the content one buffer to another buffer using GPU.

Parameter	Description
pSrcBuffer [in]	A pointer to the source *AMFBuffer* object
pDstBuffer [in]	A pointer to the destination *AMFBuffer* object
srcOffset [in]	Source offset in bytes
dstOffset [in]	Destination offset in bytes
size [in]	Size of the data to be copied in bytes

Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::CopyPlane

```
AMF_RESULT AMF_STD_CALL CopyPlane(AMFPlane *pSrcPlane, const amf_size srcOrigin[3],
       const amf_size region[3], AMFPlane *pDstPlane, const amf_size dstOrigin[3]);
```

Copy the content of a plane to another plane.

The *srcOrigin, dstOrigin* and the *region* parameters represent the 3D coordinates and the size of the area of the plane to be filled. For 2D planes *srcOrigin[2], dstOrigin[2]* and *region[2]* must be set to 0.

Parameter	Description
pSrcPlane [in]	A pointer to an AMFPlane object to be copied
pDstPlane [in]	A pointer to an AMFPlane object to be copied to
srcOrigin [in]	A triplet specifying the origin of a rectangular area in the source plane
dstOrigin [in]	A triplet specifying the origin of a rectangular area in the destination plane
region [in]	A triplet specifying the size of a rectangular area in the plane to be filled
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::CopyBufferToHost

```
AMF_RESULT AMF_STD_CALL CopyBufferToHost(AMFBuffer* pSrcBuffer, amf_size srcOffset,
       amf_size size, void* pDest, bool blocking);
```

Copy the content of a buffer from GPU memory to host (CPU) memory.

The destination buffer is not an AMF object and can be allocated using any host memory allocation methods, such as *malloc*, the *new* operator, etc. The application is responsible for freeing this buffer when it is no longer needed.

When the *blocking* parameter is set to *false*, the call to *CopyBufferToHost* returns immediately. Set a synchronization point immediately after the call to *CopyBufferToHost* to determine when the copy operation is completed.

Parameter	Description
pSrcBuffer [in]	A pointer to the source AMFBuffer object
pDst [in]	A pointer to the destination buffer in host memory.
srcOffset [in]	Source offset in bytes
size [in]	Size of the data to be copied in bytes
blocking [in]	When set to *true*, the call will block until the operation is completed. When set to *false*, the call will return immediately and the copy operation will continue in the background
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::CopyBufferFromHost

```
AMF_RESULT AMF_STD_CALL CopyBufferFromHost(const void* pSource, amf_size size,
        AMFBuffer* pDstBuffer, amf_size dstOffset, bool blocking);
```

Copy the content of a buffer from host (CPU) memory to GPU memory.

When the *blocking* parameter is set to *false*, the call to *CopyBufferToHost* returns immediately. Set a synchronization point immediately after the call to *CopyBufferToHost* to determine when the copy operation is completed.

Parameter	**Description**
pSource [in]	A pointer to the source buffer in host memory.
pDstBuffer [in]	A pointer to the destination *AMFBuffer* object
dstOffset [in]	Source offset in bytes
size [in]	Size of the data to be copied in bytes
blocking [in]	When set to *true*, the call will block until the operation is completed. When set to *false*, the call will return immediately and the copy operation will continue in the background
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::CopyPlaneToHost

```
AMF_RESULT AMF_STD_CALL CopyPlaneToHost(AMFPlane *pSrcPlane, const amf_size origin[3],
        const amf_size region[3], void* pDest, amf_size dstPitch, bool blocking);
```

Copy a 2D surface plane in GPU memory to a buffer in host (CPU) memory.

The *origin* and the *region* parameters represent the 3D coordinates and the size of the area of the plane to be filled. For 2D planes *origin[2]* and *region[2]* must be set to 0.

The destination buffer is not an AMF object and can be allocated using any host memory allocation methods, such as *malloc*, the *new* operator, etc. The application is responsible for freeing this buffer when it is no longer needed.

When the *blocking* parameter is set to *false*, the call to *CopyPlaneToHost* returns immediately. Set a synchronization point immediately after the call to *CopyPlaneToHost* to determine when the copy operation is completed.

Parameter	**Description**
pSrcPlane [in]	A pointer to an *AMFPlane* object to be copied
pDest [in]	A pointer to the destination buffer
origin [in]	A triplet specifying the origin of a rectangular area in the source plane
region [in]	A triplet specifying the size of a rectangular area in the plane to be copied
dstPitch [in]	Destination pitch (the size of a single scanline) in bytes
blocking [in]	When set to *true*, the call will block until the operation is completed. When set to *false*, the call will return immediately and the copy operation will continue in the background
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::CopyPlaneFromHost

```
AMF_RESULT AMF_STD_CALL CopyPlaneFromHost(void* pSource, const amf_size origin[3],
       const amf_size region[3], amf_size srcPitch, AMFPlane *pDstPlane, bool blocking);
```

Copy a buffer in host (CPU) memory to a 2D surface plane in GPU memory.

The *origin* and the *region* parameters represent the 3D coordinates and the size of the area of the plane to be filled. For 2D planes *origin[2]* and *region[2]* must be set to 0.

When the *blocking* parameter is set to *false*, the call to *CopyPlaneFromHost* returns immediately. Set a synchronization point immediately after the call to *CopyPlaneFromHost* to determine when the copy operation is completed.

Parameter	**Description**
pSource [in]	A pointer to the source buffer in host memory
pDstPlane [in]	A pointer to an *AMFPlane* object to copy to
origin [in]	A triplet specifying the origin of a rectangular area in the destination plane
region [in]	A triplet specifying the size of a rectangular area in the plane to be copied
srcPitch [in]	Source pitch (the size of a single scanline) in bytes
blocking [in]	When set to *true*, the call will block until the operation is completed. When set to *false*, the call will return immediately and the copy operation will continue in the background
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFCompute::ConvertPlaneToPlane

```
AMF_RESULT AMF_STD_CALL ConvertPlaneToPlane(AMFPlane* pSrcPlane, AMFPlane** ppDstPlane,
       AMF_CHANNEL_ORDER order, AMF_CHANNEL_TYPE type);
```

Copy a plane in GPU memory to another plane of a different format in GPU memory.

The order of channels is specified using the *AMF_CHANNEL_ORDER* enumeration defined as follows:

Value	Description
AMF_CHANNEL_ORDER_R	Only the Red channel is present
AMF_CHANNEL_ORDER_RG	Only the Red channel followed by the Green channel are present
AMF_CHANNEL_ORDER_BGRA	The Blue, Green, Red and Alpha channels are present
AMF_CHANNEL_ORDER_RGBA	The Red, Green, Blue and Alpha channels are present
AMF_CHANNEL_ORDER_ARGB	The Alpha, Red, Green and Blue channels are present

The channel type defines the format of pixel data in GPU memory and is defined as follows:

Value	Description
AMF_CHANNEL_UNSIGNED_INT8	Unsigned 8-bit integer per channel
AMF_CHANNEL_UNSIGNED_INT32	Unsigned 32-bit integer per channel
AMF_CHANNEL_UNORM_INT8	Unsigned normalized 8-bit integer per channel
AMF_CHANNEL_UNORM_INT16	Unsigned normalized 16-bit integer per channel
AMF_CHANNEL_SNORM_INT16	Signed normalized 16-bit integer per channel
AMF_CHANNEL_FLOAT	Floating point
AMF_CHANNEL_FLOAT16	16-bit floating point

Blah

Parameter	Description
pSrcPlane [in]	A pointer to the source *AMFPlane* object
pDstPlane [out]	A pointer to a location to receive a pointer to the destination *AMFPlane* object
order [in]	Channel order
type [in]	Pixel format
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

2.5.5 AMFComputeKernel

The *AMFComputeKernel* interface facilitates passing parameters to and execution of an AMFCompute kernel. A pointer to the *AMFComputeKernel* interface of a specific kernel can be obtained by calling the *AMFCompute::GetKernel* method.

The *AMFComputeKernel* interface inherits from *AMFInterface*.

Include *public/include/core/Compute.h*

AMFComputeKernel::GetNative

```
void* MF_STD_CALL GetNative();
```

Get a native kernel handle.

Return Value	Native kernel handle

AMFComputeKernel::GetIDName

```
const wchar_t* AMF_STD_CALL GetIDName();
```

Get the Identification Name of a kernel. This name is used to identify a kernel in the AMF Performance Trace log.

The pointer returned points to an internally allocated wide-character Unicode string. Do not free this memory when it is no longer needed. Do not save the pointer returned by *GetIDName* in any location that might outlive the *AMFComputeKernel* object itself.

Return Value	A pointer to a string identifying the kernel.

Passing Parameters to a Kernel

```
AMF_RESULT AMF_STD_CALL SetArgPlaneNative(amf_size index, void* pPlane,
       AMF_ARGUMENT_ACCESS_TYPE eAccess);
AMF_RESULT AMF_STD_CALL SetArgBufferNative(amf_size index, void* pBuffer,
       AMF_ARGUMENT_ACCESS_TYPE eAccess);
AMF_RESULT AMF_STD_CALL SetArgPlane(amf_size index, AMFPlane* pPlane,
       AMF_ARGUMENT_ACCESS_TYPE eAccess);
AMF_RESULT AMF_STD_CALL SetArgBuffer(amf_size index, AMFBuffer* pBuffer,
       AMF_ARGUMENT_ACCESS_TYPE eAccess);
AMF_RESULT AMF_STD_CALL SetArgInt32(amf_size index, amf_int32 data);
AMF_RESULT AMF_STD_CALL SetArgInt64(amf_size index, amf_int64 data);
AMF_RESULT AMF_STD_CALL SetArgFloat(amf_size index, amf_float data);
AMF_RESULT AMF_STD_CALL SetArgBlob(amf_size index, amf_size dataSize, const void* pData);
```

These methods are used to pass parameters to an AMFCompute kernel. Each parameter is identified by a zero-based index.

The access rights for the kernel are defined by the *eAccess* parameter and can be one of the following values:

Value	Description
AMF_ARGUMENT_ACCESS_READ	Read-only access
AMF_ARGUMENT_ACCESS_WRITE	Write-only access
AMF_ARGUMENT_ACCESS_READWRITE	Read and write access

Parameter	**Description**
index [in]	A zero-based parameter index
pPlane [in]	A handle of a native OpenCL plane or a pointer to an *AMFPlane* object
pBuffer [in[	A handle of a native OpenCL buffer or a pointer to an *AMFBuffer* object
data [in]	Data of a primitive type
pData [in]	A pointer to unstructured data buffer in host memory
dataSize [in]	The size of the unstructured data buffer passed in pData
eAccess [in]	Data access mode
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

AMFComputeKernel::GetCompileWorkgroupSize

```
AMF_RESULT AMF_STD_CALL GetCompileWorkgroupSize(amf_size workgroupSize[3]);
```

Get compile workgroup size.

The *GetCompileWorkgroupSize* method is a wrapper around the *clGetKernelWorkGroupInfo* OpenCL call with *CL_KERNEL_COMPILE_WORK_GROUP_SIZE* passed as parameter. The result is returned as an array of 3 elements (X,Y,Z). When the workgroup size is not specified, the returned value would be (0,0,0).

For more information about compile workgroup size please refer to the OpenCL documentation: https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clGetKernelWorkGroupInfo.html

Parameter	**Description**
workgroupSize [out]	Workgroup size (X,Y,Z)
Return Value	*AMF_OK* on success *AMF_FAIL* on failure

	AMF_INVALID_ARG when any of the arguments is invalid

AMFComputeKernel::Enqueue

```
AMF_RESULT AMF_STD_CALL Enqueue(amf_size dimension, amf_size globalOffset[3], amf_size
        globalSize[3], amf_size localSize[3]);
```

Submit a kernel for execution.

Parameter	Description
dimension [in]	The number of dimensions used to specify the global work-items and work-items in the work-group. *dimension* must be greater than zero and less than or equal to three.
globalOffset [in]	Must currently be a *NULL* value. In a future revision of AMF Compute, *globalOffset* can be used to specify an array of unsigned values that describe the offset used to calculate the global ID of a work-item instead of having the global IDs always start at offset (0, 0,… 0).
globalSize [in]	Points to an array of unsigned values that describe the number of global work-items in *dimensions* dimensions that will execute the kernel function.
localSize [in]	*localSize* will be used to determine how to break the global work-items specified by *globalSize* into appropriate work-group instances. If *localSize* is specified, the values specified in *globalSize[0],…globalSize[dimension - 1]* must be evenly divisable by the corresponding values specified in *localSize[0],…localSize[dimension - 1]*.
Return Value	*AMF_OK* on success *AMF_FAIL* on failure *AMF_INVALID_ARG* when any of the arguments is invalid

2.5.6 AMFComputeSyncPoint

A synchronization point allows the CPU to wait for the completion and query the status of a particular operation submitted to an AMF Compute queue. It is logically similar to the Win32 Event synchronization object, but is designed to synchronize a CPU with a GPU.

A synchronization point object is created when a sync point is added to the AMF Compute queue using the *AMFDeviceCompute::PutSyncPoint* method.

The *AMFComputeSyncPoint* interface inherits from *AMFInterface*.

Include *public/include/core/Compute.h*

AMFComputeSyncPoint::IsCompleted

```
bool AMF_STD_CALL IsCompleted();
```

Determine whether a synchronization point has been reached by the GPU.

This method allows the CPU to unintrusively check whether a certain set of GPU operations has been completed.

Return Value	*true* when the synchronization point has been reached, *false* otherwise.

AMFComputeSyncPoint::Wait

```
void AMF_STD_CALL Wait();
```

Block CPU execution until the synchronization point is reached by the GPU.

This method allows the CPU to wait for the completion of certain jobs without taking up any CPU cycles. It will not return and put the calling thread to sleep until the GPU reaches the set synchronization point.

2.6 Components

2.6.1 AMFComponent

The *AMFComponent* interface provides access to the functionality of an AMF component. All AMF components implement the *AMFComponent* interface.

The main purpose of an AMF component is to process a media stream, usually as part of a pipeline.

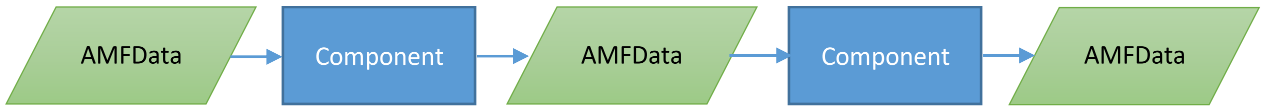

Fig. 1 – A pipeline of AMF components

The *AMFComponent* interface inherits from the *AMFPropertyStorageEx* interface. All AMF components are thread-safe.

Standard AMF components are created using the *AMFFactory::CreateComponent* method.

The use model of AMF components is built around the following flow:

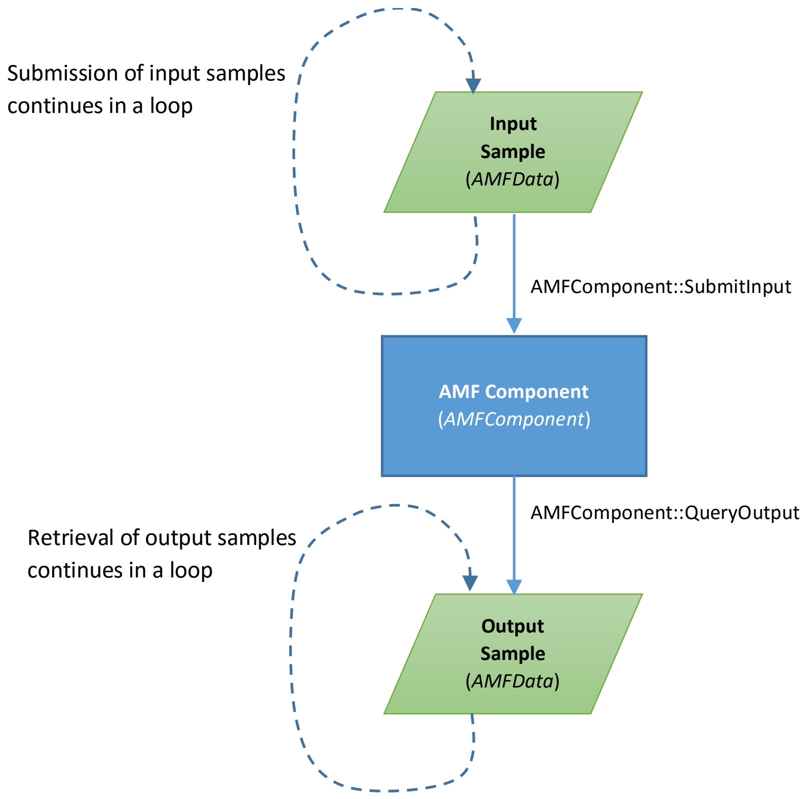
Submission of input samples continues in a loop
Input Sample (AMFData)
AMFComponent::SubmitInput
AMF Component (AMFComponent)
AMFComponent::QueryOutput
Retrieval of output samples continues in a loop
Output Sample (AMFData)

Fig. 2 – AMF Component Usage Model

Both input and output samples are stored in objects implementing the AMFData interface.

Input samples are submitted continuously to a component by calling the *SubmitInput* method. The component processes input samples and produces output samples, which are placed in the output queue. Output samples are retrieved from the output queue by continuously calling the *QueryOutput* method. Since AMF components are thread-safe, submission of input samples and retrieval of output samples can be done either from a single thread, or multiple threads.

User code should not make any assumptions about any relationship between input and output samples. While for some components the number of output samples is equal to the number of input samples, for other components this is not true. Some components may require more than one input sample to be submitted before any output samples are produced.

AMF does not provide a standard implementation of a pipeline as part of the AMF API, leaving it up to applications to implement. However, many AMF samples do include a pipeline implementation, which could be used as a basis for your own implementation.

Include *public/include/components/Component.h*

AMFComponent::Init()

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

Initialize a component. This method fully initializes the component and should be called at least once before the component can be used.

Components can be initialized multiple times with either the *Init* or the *ReInit* methods. Before a component can be initialized again with the *Init* method, it needs to be terminated by calling the *Terminate* method on the same object.

Parameter	**Description**
format [in]	Pixel format. Depending on the function of the component, this parameter may contain either the input, or the output format, or both.
width [in]	Width in pixels
height [in]	Height in scan lines
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFComponent::ReInit()

```
AMF_RESULT  AMF_STD_CALL ReInit(amf_int32 width, amf_int32 height);
```

Reinitialize a component for the new resolution. The *ReInit* method performs a minimal reinitialization and typically is much quicker than *Init*. Call *ReInit* for the fast resolution change when resolution is the only parameter that has changed.

ReInit requires that *Init* is called at least once prior to the call and will fail when this is not so.

Parameter	**Description**
width [in]	Width in pixels
height [in]	Height in scan lines
Return Value	*AMF_OK* on success

	AMF_NO_DEVICE when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid

AMFComponent::Terminate()

```
AMF_RESULT  AMF_STD_CALL Terminate();
```

Terminate a component.

Components need to be terminated before they can be initialized again with a call to the *Init* method. Component objects being reinitialized with the *ReInit* method should not call *Terminate* prior to calling *ReInit*.

Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized

AMFComponent::SubmitInput()

```
AMF_RESULT  AMF_STD_CALL SubmitInput(AMFData* pData);
```

Submit a new input sample to the *AMFComponent* object.

Depending on the component and how it has been initialized, multiple input samples might be required to produce any output samples. For example, decoders for codecs that perform frame reordering may require several samples to be submitted before decoding of the first frame can be performed. In this case the client code should continuously call *SubmitInput* until the minimum number of input samples required has been submitted.

Many components queue input samples and the input queue may have a limited depth. When the input queue is full, *SubmitInput* would return *AMF_INPUT_FULL*. When this condition occurs, further submission of input samples should be suspended until at least one output sample has been retrieved using the *QueryOutput* method. However, when the depth of the input queue of a component is known, for performance reasons it is best to avoid calling *SubmitInput* until it fails with *AMF_INPUT_FULL* and track the number of submitted input samples.

Input samples are tracked after being submitted to the component with *SubmitInput* using the *AMFBufferObserver* or *AMFSurfaceObserver* interfaces, depending on the type of input sample.

Parameter	**Description**
pData [in]	Input sample
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when *pData* is *nullptr* *AMF_INPUT_FULL* when the input queue is full

AMFComponent::QueryOutput()

```
AMF_RESULT  AMF_STD_CALL QueryOutput(AMFData** ppData);
```

Retrieve a sample from the output queue.

After an input sample has been submitted to an *AMFComponent* object, output samples are placed into the output queue where they can be retrieved from using the *QueryOutput* method. When an output sample becomes available, *QueryOutput* returns *AMF_OK* and places a pointer to the output sample into a location pointed to by the *ppData* parameter, removing the sample from the output queue. If an output sample is not available yet, *QueryOutput* will return *AMF_REPEAT*, indicating that the call needs to be retried after some period of time (note that some components might return *AMF_OK*, but *ppData* would receive a *nullptr* when the data is not available yet).

When draining has been initiated (see *AMFComponent::Drain* for more detail) and the last output sample has been retrieved, *QueryOutput* returns *AMF_EOF*. This indicates the end of the drain operation, after which input samples can continue to be submitted.

Parameter	**Description**
ppData [out]	A pointer to a location to receive a pointer to the output sample
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when *ppData* is *nullptr* *AMF_REPEAT* when the output queue is empty *AMF_EOF* when the last sample has been collected after draining the output queue

AMFComponent::Drain()

```
AMF_RESULT  AMF_STD_CALL Drain();
```

Drain all submitted input samples. Draining is used to clear the output queue witout loosing any samples that have already been produced by the *AMFComponent* object. Draining forces the object to produce output even when the object would normally require more input before output is produced. *Drain* is typically called at the end of the stream.

Client code should stop submitting new input samples to the component after *Drain* has been called and until all available output samples have been retrieved, which is indicated by the *AMF_EOF* being returned by *QueryOutput*.

Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized

AMFComponent::Flush()

```
AMF_RESULT  AMF_STD_CALL Flush();
```

Flush the *AMFComponent* object, discarding any samples already submitted or processed. Unlike *Drain*, *Flush* clears the input and the output queues immediately. *Flush* is typically called when the stream is interrupted by the seek operation or resolution change.

Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized

AMFComponent::GetContext()

```
AMFContext* AMF_STD_CALL GetContext();
```

Obtain a pointer to the AMFContext object the *AMFComponent* object is associated with.

GetContext does not increment the reference count on the AMFContext object it returns.

Return Value	A pointer to the *AMFContext* object associated with the object

AMFComponent::SetOutputDataAllocatorCB()

```
AMF_RESULT  AMF_STD_CALL SetOutputDataAllocatorCB(AMFDataAllocatorCB* callback);
```

Register a callback to provide a custom allocator for output *AMFData* objects (buffers or surfaces). Setting the callback to *nullptr* unregisters the callback.

Parameter	Description
callback [in]	A pointer to a custom allocator object implementing the AMFDataAllocatorCB interface
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized

AMFComponent::GetCaps

```
AMF_RESULT AMF_STD_CALL GetCaps(AMFCaps** ppCaps);
```

Get AMFComponent object capabilities.

The *AMFCaps* interface is an optional interface allowing the application to query component's capabilities.

Parameter	Description
ppCaps [out]	A pointer to a custom allocator object implementing the AMFCaps interface
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_NOT_SUPPORTED* when the object does not implement the *AMFCaps* interface

2.6.2 AMFCaps

The *AMFCaps* interface allows the application to query the component's capabilities on the current hardware.

Not all components are required to implement the *AMFCaps* interface. Always check the return code when calling *AMFComponent::GetCaps*.

AMFCaps inherits from AMFPropertyStorage.

Include *public/include/components/ComponentCaps.h*

AMFCaps::GetAccelerationType

```
AMF_ACCELERATION_TYPE AMF_STD_CALL GetAccelerationType() const;
```

Determine the level of hardware acceleration of the AMFComponent object on the current hardware.

Acceleration types are defined using the *AMF_ACCELERATION_TYPE* enumeration:

Value	Description
AMF_ACCEL_NOT_SUPPORTED	The component is not supported on the current hardware
AMF_ACCEL_HARDWARE	Full hardware acceleration is supported using a fixed function hardware block
AMF_ACCEL_GPU	Hardware acceleration is supported using programmable hardware (shaders)

AMF_ACCEL_CPU	Functionality is supported, but not hardware-accelerated

Return Value Acceleration type

AMFCaps::GetInputCaps
AMFCaps::GetOutputCaps

```
AMF_RESULT AMF_STD_CALL GetInputCaps(AMFIOCaps** ppCaps);
AMF_RESULT AMF_STD_CALL GetOutputCaps(AMFIOCaps** ppCaps);
```

Get input and output capabilities of a component.

Parameter	Description
ppCaps [out]	A pointer to a location to receive a pointer to the *AMFIOCaps* interface
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid or *ppBuffer* is *nullptr*

2.6.3 AMFIOCaps

The *AMFIOCaps* interface provides methods to query capabilities of a component's input and output.

AMFIOCaps inherits from *AMFInterface*.

Include *public/include/components/ComponentCaps.h*

AMFIOCaps::GetWidthRange
AMFIOCaps::GetHeightRange

```
void AMF_STD_CALL GetWidthRange(amf_int32* minValue, amf_int32* maxValue) const;
void AMF_STD_CALL GetHeightRange(amf_int32* minValue, amf_int32* maxValue) const;
```

Query the range of supported resolutions.

Parameter	Description
minValue [out]	A pointer to a location to receive the minimum value of the range of supported resolutions
maxValue [out]	A pointer to a location to receive the maximum value of the range of supported resolutions
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized

AMFIOCaps::GetVertAlign

```
amf_int32 AMF_STD_CALL GetVertAlign() const
```

Get vertical alignment of the image.

Return Value Vertical alignment in scanlines

AMFIOCaps::GetNumOfFormats

```
amf_int32 AMF_STD_CALL GetNumOfFormats() const;
```

Get the total number of pixel formats.

Return Value Total number of supported formats

AMFIOCaps::GetFormatAt

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

Query the level of support of each pixel format.

The *GetNumOfFormats* method returns the total number of pixel formats supported on either input or output.

Parameter	**Description**
Index [in]	A zero-based index of the format in the list of supported formats
format [out]	A pointer to a location to receive the format
native [out]	A pointer to a location to receive the value indicating whether color space conversion is required for a specific format
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when *format* or *native* are *nullptr*.

AMFIOCaps::GetNumOfMemoryTypes

```
amf_int32 AMF_STD_CALL GetNumOfMemoryTypes() const;
```

Get the total number of supported memory types.

Return Value	Total number of supported memory types

AMFIOCaps::GetMemoryTypeAt

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

Query the level of support of each memory type.

The *GetNumOfMemoryTypes* method returns the total number of memory types supported on either input or output.

Parameter	**Description**
Index [in]	A zero-based index of the format in the list of supported formats
memType [out]	A pointer to a location to receive the memory type
native [out]	A pointer to a location to receive the value indicating whether color space conversion is required for a specific format
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when *format* or *native* are *nullptr*.

AMFIOCaps::IsInterlacedSupported

```
amf_bool AMF_STD_CALL IsInterlacedSupported() const;
```

Check whether interlaced input or output is supported.

Return Value	*True* when interlaced content is supported, *false* otherwise

2.6.4 AMFDataAllocatorCB

The *AMFDataAllocatorCB* interface is used to facilitate interaction between an *AMFComponent* object and a custom memory allocator.

AMFDataAllocatorCB inherits from *AMFInterface*.

Include *public/include/components/Component.h*

AMFDataAllocatorCB::AllocBuffer

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

This method is called when the *AMFComponent* object requests allocation of an *AMFBuffer* object.

Parameter	**Description**
type [in]	Memory type
size [in]	Buffer size in bytes
ppBuffer [out]	A pointer to a location to receive a pointer to the newly allocated buffer
Return Value	*AMF_OK* on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type or size are invalid or *ppBuffer* is *nullptr*

AMFDataAllocatorCB::AllocSurface

```
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);
```

This method is called when the *AMFComponent* object requests allocation of an *AMFSurface* object.

Parameter	**Description**
type [in]	Memory type
format [in]	Pixel format
width [in]	Surface width in pixels
height [in]	Surface height in scan lines
hPitch [in]	Horizontal pitch in bytes
vPitch [in]	Vertical pitch is scanlines
ppSurface [out]	A pointer to the location to receive a pointer to the *AMFSurface* interface
	AMF_OK on success *AMF_NO_DEVICE* when the context hasn't been initialized *AMF_INVALID_ARG* when memory type, format or surface sizes are invalid

3 Using AMF API

A typical application workflow includes the following steps:

1. Initialie AMF runtime, obtain a pointer to the *AMFFactory* interface.
2. Create a native DirectX, OpenGL or OpenCL device using the appropriate DirectX, OpenGL or OpenCL API
3. Create an AMF context from the native device using the *AMFFactory::CreateContext* method.
4. Create the necessary AMF components using the *AMFFactory::CreateComponent* method and build an application-specific pipeline.
5. Initialize every component by calling the *AMFComponent::Init* method on each of the component object.
6. The pipeline receives samples from an external source (such as, for instance, a Webcam or a source file) and submits samples to the first component using the *AMFComponent::SubmitInput* method.
7. The pipeline retrieves output samples from the first component using the *AMFComponent::QueryOutput* method and submits them to the next component in the pipeline by passing them as a parameter to the *AMFComponent::SubmitInput* method of the next component object in the pipeline. This process is repeated for each component in the pipeline.

Currently the AMF runtime includes the following components:

1. Video Decoder, supporting all codecs supported by the underlying hardware
2. Video Encoder supporting the h.264 AVC and SVC codec
3. Video Converter performing color space conversions

Please refer to the appropriate documentation for the information on specific components.