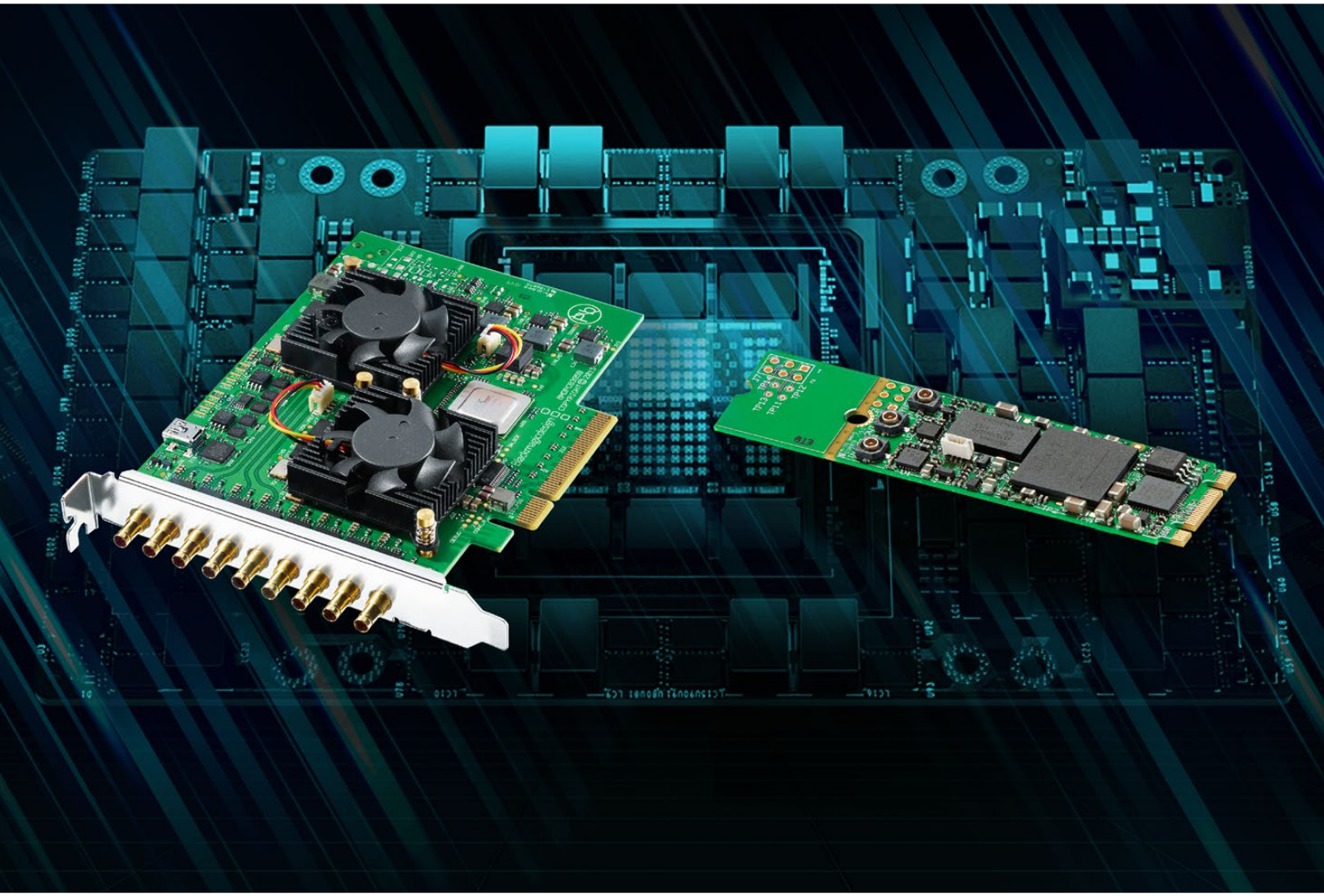
November 2024
Migration Guide

Blackmagicdesign

DeckLink SDK



DeckLink SDK Migration Guide

macOS™
Windows™
Linux™

Contents

Desktop Video 14.3 SDK Migration Guide

Overview

This guide provides a description of API changes for migrating existing software applications to Desktop Video 14.3.

Desktop Video 14.3 consists of new features to provide support of more advanced memory usage to take advantage of Apple Silicon Unified Memory. With the default allocator, the video frames will be allocated with IOSurface, rather than on the system heap. The advantage of this change is that video frame buffers can interface natively, with zero-copy, between Apple processes and frameworks such as AVFoundation and Metal and the DeckLinkAPI processes.

Software built for all operating systems will benefit from Desktop Video 14.3 in respect to input allocators. The total memory allocation size required to commence capture has been significantly reduced. In addition, the input allocator will now allocate more frames when captured frames are retained for longer by the application.

NOTE As with all Desktop Video SDK releases, all interfaces deprecated in Desktop Video 14.3 have their unique identifiers retained. This means that applications that are built with earlier Desktop Video SDK versions will continue to work against Desktop Video 14.3, via a legacy layer implemented in the DeckLinkAPI. Irrespective of the SDK version built against, all macOS applications running against Desktop Video 14.3 implemented with default allocators will take advantage of native IOSurface-based buffers.

Summary of changes

There are a number of key changes to Desktop Video 14.3 SDK.

1. The `IDeckLinkMemoryAllocator` interface has been replaced with the `IDeckLinkVideoBufferAllocatorProvider` and `IDeckLinkVideoBufferAllocator` interfaces.
2. Removed the `IDeckLinkInput::SetVideoInputFrameMemoryAllocator` method, the memory allocators are provided with the new method `IDeckLinkInput::EnableVideoInputWithAllocatorProvider`.
3. Removed the `IDeckLinkOutput::SetVideoOutputFrameMemoryAllocator` method, custom memory allocation should be implemented external to the DeckLinkSDK and generated buffers provided via the `IDeckLinkOutput::CreateVideoFrameWithBuffer` method.
4. Removed the `IDeckLinkEncoderInput::SetMemoryAllocator` method, only the default allocator can be used with `IDeckLinkEncoderInput` capture.
5. Added the new interface `IDeckLinkVideoBuffer` that provides access to the underlying video frame buffer.
6. Added the new interface `IDeckLinkMacVideoBuffer`, a specialized video buffer that provides access to the underlying Mac IOSurface via CVPixelBuffer.
7. The `IDeckLinkVideoFrame::GetBytes` method has been removed. Frame buffer access is now provided via the `IDeckLinkVideoBuffer` or `IDeckLinkMacVideoBuffer` interfaces.

8. Added a new interface `IDeckLinkMacOutput` that extends `IDeckLinkOutput` with a method to create an `IDeckLinkMutableVideoFrame` interface pointer that wraps a `CVPixelBufferRef` object.
9. Added the new helper method `IDeckLinkMutableVideoFrame::SetInterfaceProvider` to simplify the implementation of custom interfaces associated with an `IDeckLinkVideoFrame`.
10. Added the new helper method `IDeckLinkOutput::RowBytesForPixelFormat` to calculate row bytes for pixel format.
11. Added the new conversion method `IDeckLinkVideoConversion::ConvertNewFrame` that creates and converts into the destination `IDeckLinkVideoFrame`, with an optional `IDeckLinkVideoBuffer` buffer.

Accessing the video frame buffer

In most cases, applications will need to migrate from `IDeckLinkVideoFrame::GetBytes` to access the underlying video frame buffer with the `IDeckLinkVideoBuffer` interface. This change provides greater flexibility to handling buffers, which is important for synchronizing memory access between the DeckLink driver and the application. As an example, here is existing code that writes to the underlying frame buffer:

```
IDeckLinkVideoFrame* videoFrame; // For simplicity assume that videoFrame
is a created interface pointer
void* frameBuffer = nullptr;

// Access underlying frame buffer address
HRESULT result = videoFrame->GetBytes(&frameBuffer);
if (result != S_OK)
{
    // Error handling
}

// Access frameBuffer for writing
```

This should be changed to:

```
IDeckLinkVideoFrame* videoFrame; // For simplicity assume that videoFrame
is a created interface pointer
IDeckLinkVideoBuffer* videoBuffer = nullptr;
void* frameBuffer = nullptr;

HRESULT result = videoFrame->QueryInterface(IID_IDeckLinkVideoBuffer,
(void**)&videoBuffer);
if (result != S_OK)
{
    // Error handling
}

// Prepare the buffer for CPU access
result = videoBuffer->StartAccess(bmdBufferAccessWrite);
if (result != S_OK)
{
    // Error handling
}

// Access underlying frame buffer address
```

```
result = videoBuffer->GetBytes(&frameBuffer);
if (result != S_OK)
{
    // Error handling
}

// frameBuffer is valid to write to

// End the CPU access - the frameBuffer address is no longer guaranteed
result = videoBuffer->EndAccess(bmdBufferAccessWrite);
if (result != S_OK)
{
    // Error handling
}

// Release buffer
videoBuffer->Release()
```

A similar process should be used for reading a buffer, e.g. for reading a captured frame. Here, use buffer flag `bmdBufferAccessRead` in calls to `IDeckLinkVideoBuffer::StartAccess` and `IDeckLinkVideoBuffer::EndAccess`

> **NOTE** To finalize buffer access, the number of calls to `IDeckLinkVideoBuffer::EndAccess` should be equal to `IDeckLinkVideoBuffer::StartAccess` with the same buffer access flags.

Accessing the captured video frame buffer on macOS

For macOS, Desktop Video 14.3 adds native support for IOSurface buffers. Rather than using `IDeckLinkVideoBuffer`, it is more efficient to access the underlying IOSurface buffer as a `CVPixelBufferRef` object. This can be achieved by querying the `IDeckLinkMacVideoBuffer` interface from the `IDeckLinkVideoFrame` interface pointer.

The DeckLink API will provide a `CVPixelBufferRef` with the following attached properties:

- Frame dimensions (width, height, row bytes)
- Pixel format
- Color primaries, Transfer function
- Mastering display color volume, where appropriate for HDR
- Content light level, where appropriate for HDR

The following is an example of obtaining a `CVPixelBufferRef` object from a captured `IDeckLinkVideoFrame` interface pointer:

```
IDeckLinkVideoFrame* videoFrame; // For simplicity videoFrame is a valid
interface pointer
IDeckLinkMacVideoBuffer* macVideoBuffer = nullptr;
CVPixelBufferRef pixelBuffer = nullptr;
void* frameBuffer = nullptr;

HRESULT result = videoFrame->QueryInterface(IID_IDeckLinkMacVideoBuffer,
(void**)&macVideoBuffer);
if (result != S_OK)
{
    // Error handling
```

```
}

// Get buffer as CVPixelBufferRef type
result = macVideoBuffer->CreateCVPixelBufferRef((void**)&pixelBuffer);
if (result != S_OK)
{
    // Error handling
}

// CVPixelBuffer will be valid for access until CVPixelBufferRef is
released

CVPixelBufferRelease(pixelBuffer);

// Release buffer
macVideoBuffer->Release()
```

Creating a video frame from pixel buffer on macOS

For macOS, Desktop Video 14.3 SDK adds a new interface `IDeckLinkMacOutput` that extends `IDeckLinkOutput` to provide the efficient transfer of `CVPixelBufferRef` objects for playback. The call to `IDeckLinkMacOutput::CreateVideoFrameFromCVPixelBufferRef` will output an `IDeckLinkMutableVideoFrame` that wraps and retains a `CVPixelBufferRef` object. This method call will also configure the `IDeckLinkVideoFrame` and `IDeckLinkVideoFrameMetadataExtensions` interfaces based on the `CVPixelBufferRef` properties.

The following is an example of creating a `IDeckLinkMutableVideoFrame` interface pointer from a `CVPixelBufferRef`:

```
IDeckLinkOutput* deckLinkOutput;   // for simplicity deckLinkOutput is a
valid interface pointer
IDeckLinkMacOutput* deckLinkMacOutput = nullptr;
IDeckLinkMutableVideoFrame* videoFrame = nullptr;
CVPixelBufferRef pixelBuffer; // For simplicity pixelBuffer is a valid
object

// Query the IDeckLinkMacOutput interface
HRESULT hr = deckLinkOutput->QueryInterface(IID_IDeckLinkMacOutput,
(void**)&deckLinkMacOutput);
if (result != S_OK)
{
    // Error handling
}

// Create an IDeckLinkMutableVideoFrame interface pointer from a
CVPixelBufferRef
hr = deckLinkMacOutput->CreateVideoFrameFromCVPixelBufferRef((void*)
pixelBuffer, &videoFrame);
if (result != S_OK)
{
    // Error handling
}

// The IDeckLinkMutableVideoFrame interface pointer will hold onto
reference to the CVPixelBufferRef
CVPixelBufferRelease(pixelBuffer);

// Schedule IDeckLinkMutableVideoFrame for playback
```

```
hr = deckLinkOutput->ScheduleVideoFrame(videoFrame, displayTime,
displayDuration, timeScale);
if (result != S_OK)
{
    // Error handling
}

if (videoFrame)
    videoFrame->Release();

if (deckLinkMacOutput)
    deckLinkMacOutput->Release();
```

New allocator for capture

NOTE On macOS, since the native video frame buffer now uses IOSurface, including Sandboxed apps, the below information may not be required for macOS implementations.

The process for setting a custom memory allocator for capture in this version of Desktop Video has changed. Applications that implement the existing `IDeckLinkMemoryAllocator` interface for capture should be updated with the following guidance.

Implement IDeckLinkVideoBuffer

Write a new class that implements the `IDeckLinkVideoBuffer` interface with the following requirements:

- In the constructor or an initializer, pass in allocated buffer type for the required framework
- Implement all methods of the `IDeckLinkVideoBuffer` interface.
    - The DeckLinkAPI will call `IDeckLinkVideoBuffer::GetBytes` to get a CPU addressable buffer that the DeckLink can DMA to. This replaces the legacy `IDeckLinkVideoFrame::GetBytes` method.
    - The DeckLinkAPI will call `IDeckLinkVideoBuffer::StartAccess` and `IDeckLinkVideoBuffer::EndAccess` to give applications the opportunity to make the buffer CPU accessible. This is required if using unified or pinned memory where the buffer access may be shared with a GPU. The return of both the `IDeckLinkVideoBuffer::StartAccess` and `IDeckLinkVideoBuffer::EndAccess` methods can be blocked if required. Both `IDeckLinkVideoBuffer::StartAccess` and `IDeckLinkVideoBuffer::EndAccess` methods can be called multiple times, so consider keeping a lock count for the access type.
    - Implement `IUnknown` as required. The implementation of `IUnknown::QueryInterface` shall handle the query for `IID_IDeckLinkVideoBuffer`.

Implement IDeckLinkVideoBufferAllocator

Write a new class that implements the `IDeckLinkVideoBufferAllocator` interface. This class could implement the memory pool for video buffer allocations if one is available. Implementers should consider caching completed video buffers since allocating memory can be a CPU intensive task. This implementation should replace any previous implementation of `IDeckLinkMemoryAllocator` with the following differences:

- The implementation of `IDeckLinkVideoBufferAllocator::AllocateVideoBuffer` should replace the existing `IDeckLinkMemoryAllocator::AllocateBuffer`, however it should output a new or cached `IDeckLinkVideoBuffer` interface pointer instead of a `void*` pointer. The existing `IDeckLinkMemoryAllocator::AllocateBuffer` method had a size input parameter, this is now provided to the `IDeckLinkVideoBufferAllocatorProvider::GetVideoBufferAllocator` with details below.
- There is no direct replacement for the `IDeckLinkMemoryAllocator::ReleaseBuffer` method, as the `IDeckLinkVideoBuffer` interface pointer should be released through reference counting via an `IUnknown` interface.
- There are no direct replacements for the `IDeckLinkMemoryAllocator::Commit` and `IDeckLinkMemoryAllocator::Decommit` methods. This functionality should be provided in the constructor and destructor of the implemented `IDeckLinkVideoBufferAllocator`, respectively, without further direction from the DeckLinkAPI.
- The `IUnknown` implementation should handle the query for `IID_IDeckLinkVideoBufferAllocator` instead of `IDeckLinkMemoryAllocator`.

Implement IDeckLinkVideoBufferAllocatorProvider

Write a new class that implements the `IDeckLinkVideoBufferAllocatorProvider` interface.

This is a key difference from the existing `IDeckLinkMemoryAllocator` interface. Previously the size parameter provided to the `IDeckLinkMemoryAllocator::AllocateBuffer` method would represented the largest buffer size required for the enabled video mode. Providing this parameter as an input to `IDeckLinkVideoBufferAllocatorProvider::GetVideoBufferAllocator` allows the memory buffer size to be known in advance, which may be required for some allocators to partition memory. Furthermore, having known width, height, rowBytes and pixel format parameters may be required by some allocators for efficient transfers into the GPU 2D space.

In addition, in the existing `IDeckLinkMemoryAllocator` implementation, the capture buffer size would also include space for ancillary data - this is now managed in its own CPU accessible buffer outside the allocated capture buffer represented by `IDeckLinkVideoBuffer`.

The base `IDeckLinkVideoBufferAllocatorProvider` interface allows for management of cases where multiple allocators are required during capture. A typical case would be for automatic mode detection, where the new detected mode is enabled with a new buffer allocation size. However, there may be still `IDeckLinkVideoBuffer` interfaces in reference that were allocated when the input was enabled in the old video mode.

To add, create a class that implements the `IDeckLinkVideoBufferAllocatorProvider` interface with the following guidance:

- Implement the `IDeckLinkVideoBufferAllocatorProvider::GetVideoBufferAllocator` method to get the implemented `IDeckLinkVideoBufferAllocator`. Use the input parameters where required by the allocator instantiation.
- Implement the `IUnknown` methods. The implementation of `IUnknown::QueryInterface` shall handle the query for `IID_IDeckLinkVideoBufferAllocatorProvider`.

A created `IDeckLinkVideoBufferAllocatorProvider` interface pointer can be passed to the DeckLinkAPI via `IDeckLinkInput::EnableVideoInputWithAllocatorProvider`.

The following is an example of implementing this new allocator scheme targeting Windows Media Foundation.

```
class MediaFoundationBuffer : public IDeckLinkVideoBuffer
{
public:
    explicit MediaFoundationBuffer(CComPtr<IMFMediaBuffer>& mediaBuffer) :
        m_refCount(1),
        m_media2DBuffer(mediaBuffer),
        m_scanLine(nullptr),
        m_lockCount(0)
    {
    }

    // IDeckLinkVideoBuffer implementation
    HRESULT GetBytes(void** buffer) override
    {
        if (!buffer)
            return E_POINTER;

        if (!m_scanLine)
            return E_ACCESSDENIED;

        *buffer = (void*)m_scanLine;
        return S_OK;
    }

    HRESULT StartAccess(BMDBufferAccessFlags flags) override
    {
        // Check whether buffer is already locked
        if (m_lockCount++ > 0)
            return S_OK;

        return m_media2DBuffer->Lock2D(&m_scanLine, nullptr);
    }

    HRESULT EndAccess(BMDBufferAccessFlags flags) override
    {
        // Keep locked until count is zero
        if (--m_lockCount > 0)
            return S_OK;

        if (m_media2DBuffer->Unlock2D() != S_OK)
            return E_FAIL;

        m_scanLine = nullptr;
        return S_OK;
    }

    // IUnknown implementation - hidden from example
    HRESULT QueryInterface(REFIID iid, LPVOID *ppv) override;
    ULONG   AddRef(void) override;
    ULONG   Release(void) override;

private:
    std::atomic<ULONG>      m_refCount;
    CComQIPtr<IMF2DBuffer>  m_media2DBuffer;
    BYTE*                   m_scanLine;
    ULONG                   m_lockCount;
};
class MediaFoundationBufferAllocator : public
IDeckLinkVideoBufferAllocator
{
public:
    explicit MediaFoundationBufferAllocator(uint32_t bufferSize) :
        m_refCount(1),
        m_bufferSize(bufferSize)
```

```
    {
    }

    // IDeckLinkVideoBufferAllocator implementation
    HRESULT AllocateVideoBuffer(IDeckLinkVideoBuffer** allocatedBuffer)
override
    {
        CComPtr<IMFMediaBuffer>         mediaBuffer;
        CComPtr<MediaFoundationBuffer>  mediaFoundationBuffer;

        if (!allocatedBuffer)
            return E_POINTER;

        // Create memory buffer aligned to 16-byte boundary
        if (MFCreateAlignedMemoryBuffer(m_bufferSize, MF_16_BYTE_
ALIGNMENT, &mediaBuffer) != S_OK)
            return E_OUTOFMEMORY;

        mediaFoundationBuffer = new MediaFoundationBuffer(mediaBuffer);

        *allocatedBuffer = mediaFoundationBuffer.Detach();
        return S_OK;
    }

    // IUnknown implementation - hidden from example
    HRESULT QueryInterface(REFIID iid, LPVOID *ppv) override;
    ULONG   AddRef(void) override;
    ULONG   Release(void) override;

private:
    std::atomic<ULONG>  m_refCount;
    uint32_t            m_bufferSize
};
```

```
class MediaFoundationVideoBufferAllocatorProvider : public
IDeckLinkVideoBufferAllocatorProvider
{
public:
    MediaFoundationVideoBufferAllocatorProvider() :
        m_refCount(1)
    {
    }

    // IDeckLinkVideoBufferAllocatorProvider implementation
    HRESULT GetVideoBufferAllocator(uint32_t bufferSize,
                                    uint32_t width,
                                    uint32_t height,
                                    uint32_t rowBytes,
                                    BMDPixelFormat pixelFormat,
                                    IDeckLinkVideoBufferAllocator**
allocator) override
    {
        CComPtr<MediaFoundationBufferAllocator>
mediaFoundationBufferAllocator;

        if (!allocator)
            return E_POINTER;

        mediaFoundationBufferAllocator = new
MediaFoundationBufferAllocator(bufferSize);
```

```
        *allocator = mediaFoundationBufferAllocator.Detach();
        return S_OK;
    }

    // IUnknown implementation - hidden from example
    HRESULT QueryInterface(REFIID iid, LPVOID *ppv) override;
    ULONG   AddRef(void) override;
    ULONG   Release(void) override;

private:
    std::atomic<ULONG>  m_refCount;
};
```

With the example above, an object of type `MediaFoundationVideoBufferAllocatorProvider` can be passed as input to `IDeckLinkInput::EnableVideoInputWithAllocatorProvider`

Custom video buffers for playback

Desktop Video 14.3 SDK provides more freedom to create custom video frame buffers without needing to register a custom memory buffer, previously provided by the `IDeckLinkOutput::SetVideoOutputFrameMemoryAllocator` method.

For playback it is often the case that a frame buffer is either provided by the video pipeline or available via a memory pool external to the DeckLinkAPI. In these cases the application can wrap the existing buffer in a class that implements the `IDeckLinkVideoBuffer` interface.

With an implemented `IDeckLinkVideoBuffer` interface, the application can then create an `IDeckLinkVideoFrame` interface pointer for scheduled playback by calling the `IDeckLinkOutput::CreateVideoFrameWithBuffer` method.

Alternatively, it is still possible to wrap a video frame object by implementing the `IDeckLinkVideoFrame` interface. In those cases, the codebase will need to be migrated to also implement the `IDeckLinkVideoBuffer` class with the following advice:

- In most cases the implementation of `IDeckLinkVideoFrame::GetBytes` can be simply migrated to `IDeckLinkVideoBuffer::GetBytes`.
- Implement both `IDeckLinkVideoBuffer::StartAccess` and `IDeckLinkVideoBuffer::EndAccess` to permit CPU access of the memory buffer in accordance with the underlying buffer.
- The implementation of `IUnknown::QueryInterface` should handle the cases of REFIID equal to either `IID_IDeckLinkVideoFrame` or `IID_IDeckLinkVideoBuffer`.

Custom video buffers for input encoder

Developers of applications that use the `IDeckLinkInputEncoder` for H.265 encoding should be aware that the new memory allocator API is not applicable for this interface. Because of this, the method `IDeckLinkEncoderInput::SetMemoryAllocator` has been removed without replacement. The `IDeckLinkInputEncoder` will capture into buffers provided by the DeckLinkAPI.

Setting HDR metadata

Previously, to set colorspace or HDR metadata associated with an output video frame, an application would need to create a custom class that implemented both the `IDeckLinkVideoFrame` and `IDeckLinkVideoFrameMetadataExtensions` interfaces. Desktop Video 14.3 SDK simplifies this process by adding the `IDeckLinkVideoFrameMutableMetadataExtensions` interface. An `IDeckLinkVideoFrameMutableMetadataExtensions` interface can be queried from an `IDeckLinkVideoFrame` or `IDeckLinkMutableVideoFrame` interface by calling `IUnknown::QueryInterface` with `IID_IDeckLinkVideoFrameMutableMetadataExtensions`.

The following is an example of creating an output frame with BT.2100 (Rec.2020 and HLG):

```
IDeckLinkOutput* deckLinkOutput;   // for simplicity deckLinkOutput is a
valid interface pointer
uint32_t frameWidth, frameHeight, frameRowBytes;  // for simplicity
variables are defined
BMDPixelFormat framePixelFormat;   // for simplicity framePixelFormat is
defined

IDeckLinkMutableVideoFrame* outputFrame = nullptr;
hr = deckLinkOutput->CreateVideoFrame(frameWidth, frameHeight,
frameRowBytes, framePixelFormat, bmdFrameFlagDefault, &outputFrame);
if (hr != S_OK)
{
    // Error handling
}

IDeckLinkVideoFrameMutableMetadataExtensions* metadataExtensions =
nullptr;
hr = outputFrame->QueryInterface(IID_
IDeckLinkVideoFrameMutableMetadataExtensions,
(void**)&metadataExtensions);
if (hr != S_OK)
{
    // Error handling
}

// Set colorspace to Rec.2020
hr = metadataExtensions->SetInt(bmdDeckLinkFrameMetadataColorspace,
bmdColorspaceRec2020);
if (hr != S_OK)
{
    // Error handling
}

// Set EOTF to HLG
hr = metadataExtensions->SetInt(bmdDeckLinkFrameMetadataHDRElectroOpticalT
ransferFunc, 3);  // Refer to CEA 861.3 for EOTF values
if (hr != S_OK)
{
    // Error handling
}

metadataExtensions->Release();

// Schedule HDR video frame for output

outputFrame->Release();
```

Providing custom interfaces to video frames

Desktop Video 14.3 SDK adds a new scheme for providing the following custom interfaces to an `IDeckLinkMutableVideoFrame` interface:

- IDeckLinkVideoFrame3DExtensions
- IDeckLinkVideoFrameMetadataExtensions
- IDeckLinkVideoFrameMutableMetadataExtensions
- IDeckLinkMacVideoBuffer
- Other classes that implement buffers from custom memory allocations

Associating the custom interface to the video frame is achieved by setting a provider with the new method `IDeckLinkMutableVideoFrame::SetInterfaceProvider`. The provider object is able to provide access to the custom interface when its `IUnknown::QueryInterface` method is called. A provider should be used here, rather than the custom interface itself, to avoid circular references.

This new method simplifies defining these custom interfaces, as it is no longer necessary to also create a wrapper implementation of the `IDeckLinkVideoFrame`/ `IDeckLinkMutableVideoFrame` interface. As an example, consider the definition of a custom class that defines a 3D video frame by implementing `IDeckLinkVideoFrame3DExtensions`

```
class VideoFrame3D : public IDeckLinkMutableVideoFrame, public
IDeckLinkVideoFrame3DExtensions
{
public:
    VideoFrame3D(com_ptr<IDeckLinkMutableVideoFrame>& left, com_
ptr<IDeckLinkMutableVideoFrame>& right);
     ~VideoFrame3D() = default;

    // IUnknown methods
    HRESULT         QueryInterface(REFIID iid, LPVOID* ppv) override;
    ULONG           AddRef(void) override;
    ULONG           Release(void) override;

    // IDeckLinkVideoFrame methods - These implement wrappers for member
m_frameLeft
    long            GetWidth(void) override;
    long            GetHeight(void) override;
    long            GetRowBytes(void) override;
    BMDPixelFormat  GetPixelFormat(void) override;
    BMDFrameFlags   GetFlags(void) override;
    HRESULT         GetBytes(void** buffer) override;
    HRESULT         GetTimecode(BMDTimecodeFormat format,
IDeckLinkTimecode **timecode) override;
    HRESULT         GetAncillaryData (IDeckLinkVideoFrameAncillary**
ancillary) override;

    // IDeckLinkMutableVideoFrame methods - These implement wrappers for
member m_frameLeft
    HRESULT         SetFlags(BMDFrameFlags newFlags) override;
    HRESULT         SetTimecode(BMDTimecodeFormat format,
IDeckLinkTimecode* timecode) override;
    HRESULT         SetTimecodeFromComponents(BMDTimecodeFormat format,
uint8_t hours, uint8_t minutes, uint8_t seconds, uint8_t frames,
BMDTimecodeFlags flags) override;
    HRESULT         SetAncillaryData(IDeckLinkVideoFrameAncillary*
ancillary) override;
    HRESULT         SetTimecodeUserBits(BMDTimecodeFormat format,
BMDTimecodeUserBits userBits) override;
```

```
    // IDeckLinkVideoFrame3DExtensions methods
    BMDVideo3DPackingFormat     Get3DPackingFormat(void) override;
    HRESULT                     GetFrameForRightEye(IDeckLinkVideoFrame**
rightEyeFrame) override;

private:
    com_ptr<IDeckLinkMutableVideoFrame>     m_frameLeft;
    com_ptr<IDeckLinkMutableVideoFrame>     m_frameRight;
    std::atomic<ULONG>                      m_refCount;
};
```

The above can be simplified to the following class definition:

```
class VideoFrame3DExtensions : public IDeckLinkVideoFrame3DExtensions
{
public:
    // IUnknown methods
    HRESULT     QueryInterface(REFIID iid, LPVOID* ppv) override;
    ULONG       AddRef(void) override;
    ULONG       Release(void) override;

    // IDeckLinkVideoFrame3DExtensions methods
    BMDVideo3DPackingFormat     Get3DPackingFormat(void) override;
    HRESULT                     GetFrameForRightEye(IDeckLinkVideoFrame**
rightEyeFrame) override;

    // Provider for the VideoFrame3DExtensions object. The provider
provides a symmetrical QueryInterface
    // between the VideoFrame3DExtensions object and its parent
IDeckLinkMutableVideoFrame object, without
    // creating a circular reference that would result in a memory leak.
    class Provider : public IUnknown
    {
    public:
        Provider(IDeckLinkMutableVideoFrame* parent, com_
ptr<IDeckLinkMutableVideoFrame>& right)

        HRESULT     QueryInterface(REFIID iid, LPVOID* ppv) override;
        ULONG       AddRef(void) override;
        ULONG       Release(void) override;

    private:
        // Note: the parent frame does not use smart pointer to avoid
circular references.
        IDeckLinkMutableVideoFrame*             m_parentFrame;
        com_ptr<IDeckLinkMutableVideoFrame>     m_rightFrame;
        std::atomic<ULONG>                      m_refCount;
    };

private:
    // Note: the VideoFrame3DExtensions constructor is private and can
only be created via the provider.
    VideoFrame3DExtensions(IDeckLinkMutableVideoFrame* owner, com_
ptr<IDeckLinkMutableVideoFrame>& right);

    com_ptr<IDeckLinkMutableVideoFrame>     m_frameLeft;
    com_ptr<IDeckLinkMutableVideoFrame>     m_frameRight;
    std::atomic<ULONG>                      m_refCount;
};
```

The implementation of the provider's QueryInterface shall access the underlying `VideoFrame3DExtensions::QueryInterface` as follows:

```
HRESULT VideoFrame3DExtensions::Provider::QueryInterface(REFIID iid,
LPVOID* ppv)
{
    com_ptr<VideoFrame3DExtensions> videoFrame3DExtensions = make_com_
ptr<VideoFrame3DExtensions>(m_parentFrame, m_rightFrame);
    return videoFrame3DExtensions->QueryInterface(iid, ppv);
}
```

The 3D extensions class can then be associated with the left-eye frame via its provider as follows:

```
com_ptr<IDeckLinkMutableVideoFrame> leftFrame;
com_ptr<IDeckLinkMutableVideoFrame> rightFrame;

// Create leftFrame + rightFrame with IDeckLinkOutput::CreateVideoFrame or
IDeckLinkOutput::CreateVideoFrameWithBuffer

com_ptr<VideoFrame3DExtensions::Provider> videoFrame3DExtensionsProvider
= make_com_ptr<VideoFrame3DExtensions::Provider>(leftFrame.get(),
rightFrame);
if (!videoFrame3DExtensionsProvider)
{
    // Error handling
}

HRESULT hr = leftFrame->SetInterfaceProvider(IID_
IDeckLinkVideoFrame3DExtensions, videoFrame3DExtensionsProvider.get());
if (hr != S_OK)
{
    // Error handling
}

// Schedule 3D output frame
```

Output video frame row bytes

Desktop Video 14.3 SDK adds a new method `IDeckLinkOutput::RowBytesForPixelFormat` for convenience when determining the required row bytes used in creating a video frame. Previously this information was only available from the `BMDPixelFormat` definition in the DeckLink SDK Manual.

The following is an example where `IDeckLinkOutput::RowBytesForPixelFormat` is useful.

```
IDeckLinkOutput* deckLinkOutput;   // for simplicity deckLinkOutput is a
valid interface pointer
uint32_t frameWidth, frameHeight;  // for simplicity frameWidth and
frameHeight are defined
BMDPixelFormat framePixelFormat;   // for simplicity framePixelFormat is
defined

uint32_t frameRowBytes;
HRESULT hr = deckLinkOutput->RowBytesForPixelFormat(framePixelFormat,
```

```
frameWidth, &frameRowBytes);
if (hr != S_OK)
{
    // Error handling
}

IDeckLinkMutableVideoFrame* outputFrame = nullptr;
hr = deckLinkOutput->CreateVideoFrame(frameWidth, frameHeight,
frameRowBytes, framePixelFormat, bmdFrameFlagDefault, &outputFrame);
if (hr != S_OK)
{
    // Error handling
}
```

Video buffers for pixel format conversion

Desktop Video 14.3 SDK adds a new conversion method `IDeckLinkVideoConversion::ConvertNewFrame` that converts to a new `IDeckLinkVideoFrame` interface pointer without the need to first create the destination video frame. The new method has input parameters for the destination pixel format and colorspace. It is also possible to provide a destination `IDeckLinkVideoBuffer` interface pointer if memory allocation is being managed externally to the DeckLinkAPI. If the buffer parameter is set to `nullptr`, the DeckLinkAPI will allocate the buffer.

When the colorspace of the converted destination frame is the same as the source frame, a value `bmdColorspaceUnknown` can be used as the colorspace parameter for `IDeckLinkVideoConversion::ConvertNewFrame`.

The following is an example for converting to a new frame with BGRA pixel format using the existing interfaces:

```
IDeckLinkVideoFrame* sourceVideoFrame;   // for simplicity
sourceVideoFrame is a valid interface pointer
IDeckLinkVideoFrame* convertedVideoFrame = nullptr;
IDeckLinkVideoConversion* videoFrameConverter = nullptr;

// Create a new video frame to convert to, matching the frame size of the
source frame
HRESULT hr = deckLinkOutput->CreateVideoFrame(sourceVideoFrame-
>GetWidth(), sourceVideoFrame->GetHeight(), sourceVideoFrame->GetWidth() *
4, bmdFormat8BitBGRA, bmdFrameFlagDefault, &convertedVideoFrame);
if (hr != S_OK)
{
    // Error handling
}

// If destination frame is Rec.2020 additional steps are required:
// Prior to Desktop Video 14.3, convertedVideoFrame would need to
be wrapped with a custom class implementing IDeckLinkVideoFrame and
IDeckLinkVideoFrameMetadataExtensions
// In Desktop Video 14.3, this process is simplified by querying the
destination video frame's IDeckLinkVideoFrameMutableMetadataExtensions
interface

// Create an IDeckLinkVideoConversion object. For Windows call
CoCreateInstance
videoFrameConverter = CreateVideoConversionInstance();
if (! videoFrameConverter)
{
```

```
    // Error handling
}

// Convert to destination frame
hr = videoFrameConverter->ConvertFrame(sourceVideoFrame,
convertedVideoFrame);
if (hr != S_OK)
{
    // Error handling
}

// Use the converted video frame

convertedVideoFrame->Release();
videoFrameConverter->Release();
```

This can be simplified with `IDeckLinkVideoConversion::ConvertNewFrame`:

```
IDeckLinkVideoFrame* sourceVideoFrame;   // for simplicity
sourceVideoFrame is a valid interface pointer
IDeckLinkVideoFrame* convertedVideoFrame = nullptr;
IDeckLinkVideoConversion* videoFrameConverter = nullptr;

// Create an IDeckLinkVideoConversion object. For Windows call
CoCreateInstance
videoFrameConverter = CreateVideoConversionInstance();
if (! videoFrameConverter)
{
    // Error handling
}

// Convert to new frame with same colorspace and default allocated
IDeckLinkVideoBuffer
hr = videoFrameConverter->ConvertNewFrame(sourceVideoFrame,
bmdFormat8BitBGRA, bmdColorspaceUnknown, nullptr, &convertedVideoFrame);
if (hr != S_OK)
{
    // Error handling
}

// Use the converted video frame

convertedVideoFrame->Release();
videoFrameConverter->Release();
```

Advice for queryable interfaces

Desktop Video 14.3 implements many related interfaces, such as `IDeckLinkVideoFrame` and `IDeckLinkVideoBuffer`. Using `IUnknown::QueryInterface` is the only supported method of getting COM interfaces. Using `dynamic_cast` for casting cannot be guaranteed to work or to continue to work across Desktop Video updates.

For example, when accessing `IDeckLinkVideoBuffer` from an `IDeckLinkVideoFrame` interface pointer:

```
IDeckLinkVideoFrame* deckLinkVideoFrame;   // for simplicity
deckLinkVideoFrame is a valid interface pointer
IDeckLinkVideoBuffer* deckLinkVideoBuffer = nullptr;

// Wrong way - potential for an invalid interface pointer
deckLinkVideoBuffer = dynamic_
cast<IDeckLinkVideoBuffer*>(deckLinkVideoFrame)

// Correct method
HRESULT hr = deckLinkVideoFrame->QueryInterface(IID_IDeckLinkVideoBuffer,
(void**)deckLinkVideoBuffer);
if (hr != S_OK)
{
    // Error handling
}
```