OpenMAX and Symbian OS - using the OpenMAX Integration Layer

Security Classification:	<External-Symbian>	**Team/Department:**	<SWE/Multimedia>
Document Reference:	SGL.TS001.601	**Author(s):**	Kevin Butchart Viviana Dudau
Status:	<Issued>	**Owner(s):**	Brian Evans
Version:	<1.0>	**Approver(s)**	
Last Revised Date:	<26><April><2006>		

Contents

Table of figures

1 Scope

The purpose of this document is showing how the **OpenMAX Integration Layer (IL) API** can be used within the **Symbian Media Device Framework** to enable access to multimedia acceleration on **Symbian OS Devices**.

The intended audience is system integrators who wish to integrate **OpenMAX IL** provided codecs into the **Symbian OS** platform. For detailed information upon the Symbian OS specific APIs mentioned in this document please consult the relevant Symbian OS platform SDK.

2 Introduction

Symbian OS provides a rich multimedia platform. This includes a client application level Multimedia **Framework (MMF)** and the hardware abstraction **Media Device Framework (MDF)**.

This paper describes the integration of **OpenMAX IL 1.0** components within the **Symbian MDF**. This primarily concentrates on integrating audio codecs within the **Symbian Multimedia DevSound** architecture and the video codecs within the **Symbian Multimedia DevVideo.**

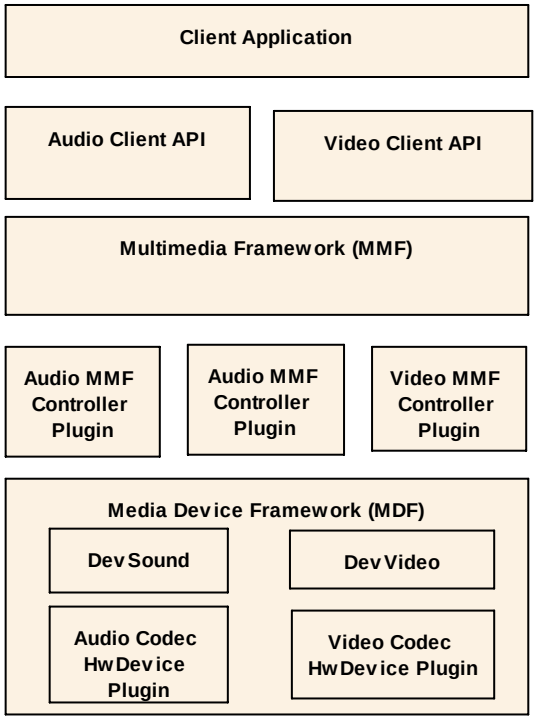


Figure 1 - Symbian Multimedia Framework

3 Audio IL Integration

3.1 Audio Overview

The audio hardware abstraction API for codecs is provided by the **CMMFHwDevice** class. This provides plug-ins to support different Codecs to the DevSound Implementation.

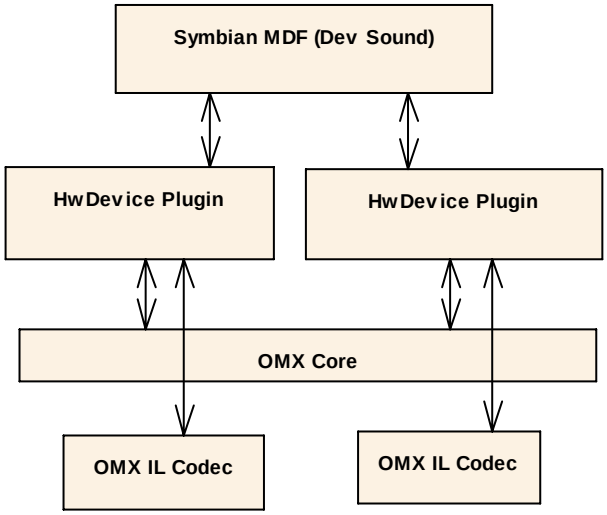


Figure 2 - MDF Architecture

Figure 2 shows the **MDF** architecture, and the integration of **OpenMAX** codecs into the HwDevice plugins. The **OpenMAX IL** client code to load and configure the desired **OpenMAX** codec components is placed in the HwDevice plugin. A codec could be implemented though a single **OpenMAX** component, or the processing involved could be implemented by a chain of components.

3.2 Audio HwDevice Plugins

These are ECom plugins that implement the **CMMFHwDevice** - the standard interface through which codec services can be provided to a **DevSound** Implementation.

3.2.1 Audio HwDevice API

To create an HwDevice plugin, one should implement the **CMMFHwDevice** interface and provide an ECom resource file to advertise the new implementation to the DevSound.

The code below is a schematic presentation of what such plugin could look like:

```
class COmxMMFHwDevice : public CMMFHwDevice
{
public:
        static COmxMMFHwDevice* NewL()
        TInt Start(TDeviceFunc aFuncCmd, TDeviceFlow aFlowCmd);
        TInt Stop();
        TInt Pause();
```

```
        TInt Init(THwDeviceInitParams& aDevInfo);
        TAny* CustomInterface(TUid aInterfaceId);
        TInt ThisHwBufferFilled(CMMFBuffer& aFillBufferPtr);
        TInt ThisHwBufferEmptied(CMMFBuffer& aEmptyBufferPtr);
        TInt SetConfig(TTaskConfig& aConfig);
        TInt StopAndDeleteCodec()
        TInt DeleteCodec()
         ~COmxMMFHwDevice();
private:
        OMX_HANDLETYPE iOmxHandle;
};
```

Table 1 **Mapping of the main API calls to their OpenMAX IL equivalents.**

CMMFHwDevice interface implementation	**OMX Methods**
NewL();	**OMX_GetHandle**(…);
Start(…);	**OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StateExecuting);
Stop();	**OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StateIdle);
Pause();	**OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StatePause);
Init(…);	**OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StateIdle);
CustomInterface(…);	**N/A –** Custom command functionality is undefined, although would generally use Get/SetParam() or Get/SetConfig() to communicate with the IL component
ThisHwBufferFilled(…);	**OMX_EmptyThisBuffer**(iOmxHandle, pBuffer);
ThisHwBufferEmptied(…);	**OMX_FillThisBuffer**(iOmxHandle, pBuffer);
SetConfig(…);	**OMX_SetParameter**(…);
StopAndDeleteCodec();	**OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StateIdle); **OMX_FreeHandle**(…);
DeleteCodec();	**OMX_FreeHandle**(…);
~**CMMFHwDevice**();	**OMX_FreeHandle**(…); if handle exists

Table 1 - API Mapping Table

3.2.2 Call Sequences

The UML notation used in all call sequences in the document is explained in section 6.3.

Figure 3 presents the call sequence for the creation of a new instance of the **CMMFHwDevice, COmxMMFHwDevice**. The new plugin talks with an OMX component, iOMXComponent.

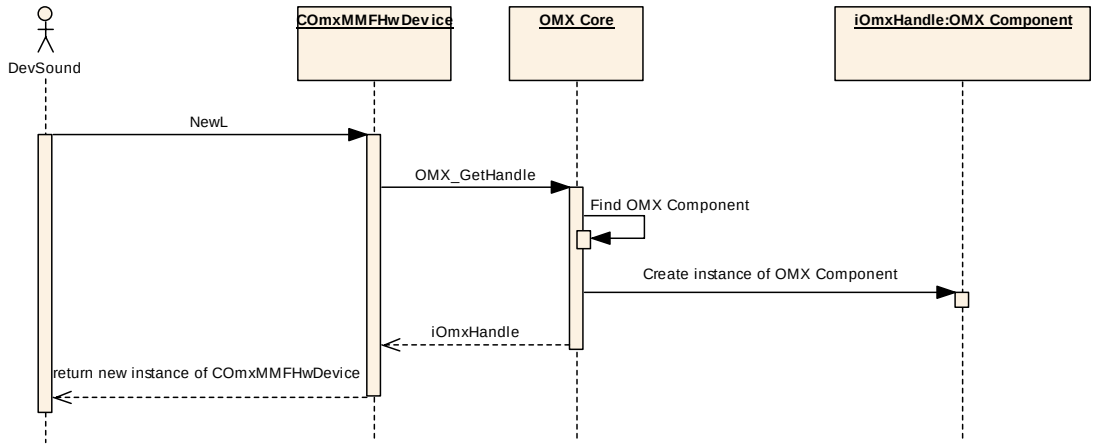


Figure 3 – Creation of a new CMMFHwDevice

Once a **CMMFHwDevice** is successfully created and it has a handle to the required **OMX** component, it can be configured via SetConfig(). The OMX component is configured using the GetParameter(…)/SetParameter(…). When all the parameters are set up the **CMMFHwDevice** will ask the **OMX** component to change its state to OMX_StateIdle (Figure 4).

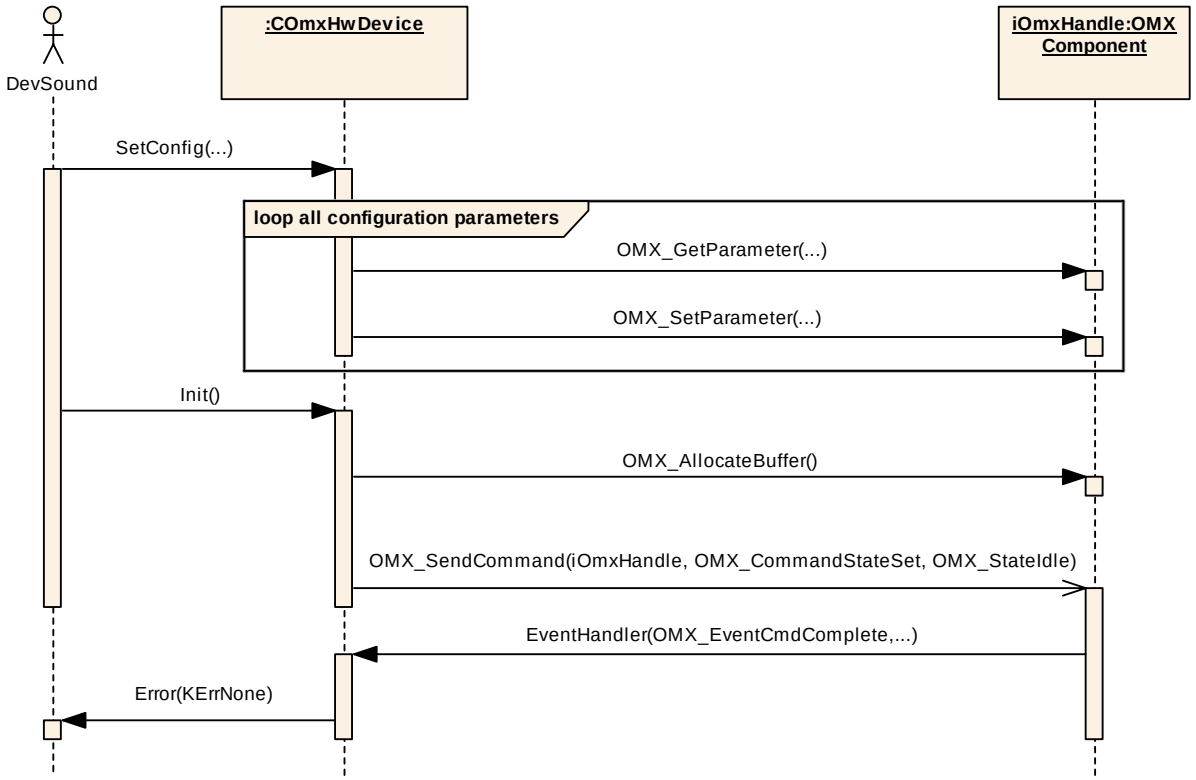


Figure 4 – Initialization of the CMMFHwDevice and OMX Component

If the initialization was successful the HwDevice can start to pass data to the OMX Component. Figure 5 shows the sequence of calls for Start() and the buffer exchange between the DevVideo, HwDevice and OMX component, while Figure 6 shows the sequence of calls for Stop().

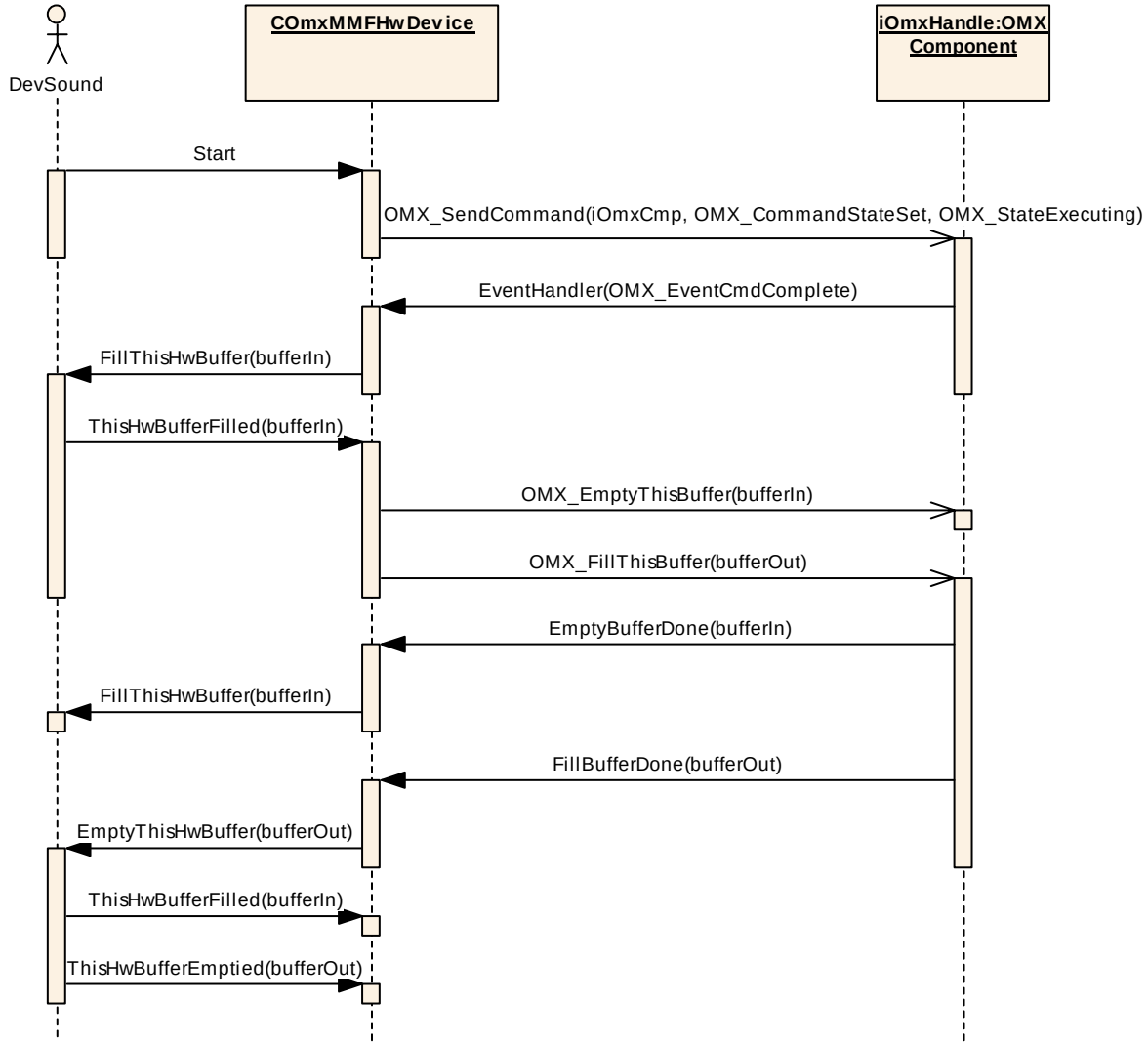


Figure 5 – Play sequence for the CMMFHwDevice and OMX Component

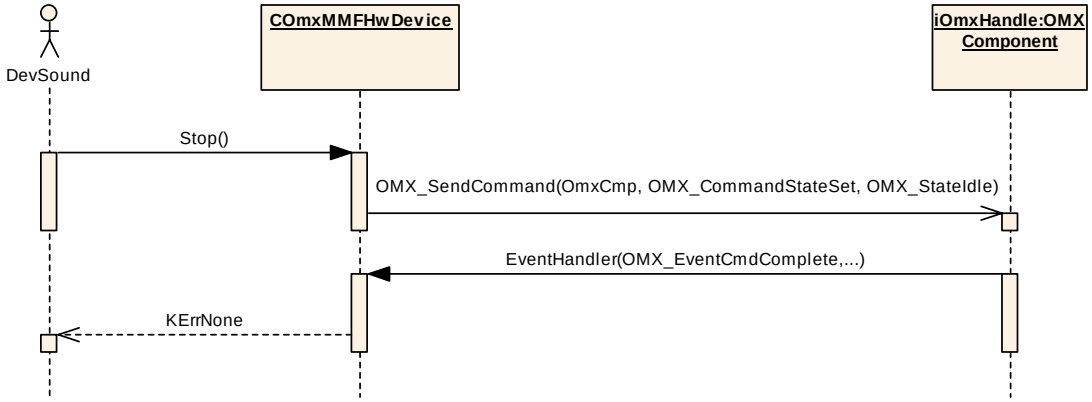


Figure 6 – Stop sequence for the CMMFHwDevice and OMX Component

3.3 Buffer Management

The buffers used in the **CMMFHwDevice** are of type **CMMFBuffer.** The HwDevice can either create the Buffers to pass back to the DevSound implementation itself, or request the IL component to create them for it.

3.3.1 IL Allocated Buffers

The HwDevice calls AllocateBuffer() within the component. It then creates a pointer buffer (**CMMFPtrBuffer**) to wrap the OMX buffer header. The **pAppPrivate** pointer in **OMX_BUFFERHEADERTYPE** can be used to store the pointer to the **CMMFPtrBuffer** for callbacks. There is no corresponding pointer in the **CMMFBuffer** structure, so a list of Buffer headers needs to be maintained with the corresponding **CMMFBuffer** within the HwDevice.

3.3.2 HwDevice Allocated Buffers

In this case the HwDevice either allocates the buffers itself or uses buffers obtained from another source (e.g. the sound driver). A call to **UseBuffer()** on the IL component will allocate a **OMX_BUFFERHEADERTYPE** structure for this buffer. Once again, the pointer back to the **CMMFBuffer** can be stored in the **pAppPrivate** structure.

4 Video IL Integration

4.1 Video Overview

The Video support in the **Symbian Media Device Framework** is provided by the **DevVideo Play** and **DevVideo Record** interfaces.

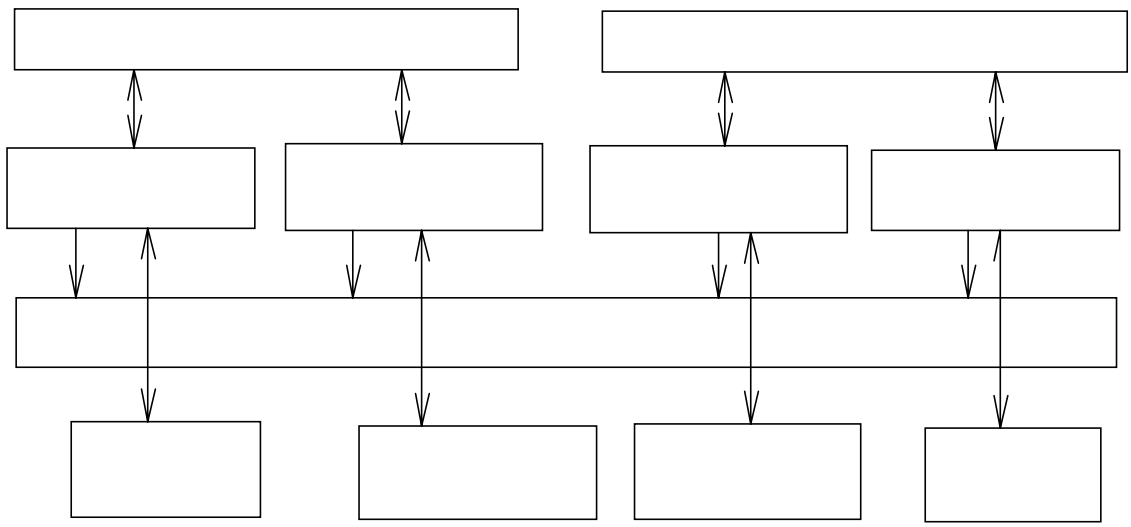


Figure 7 – DevVideo Structure

4.2 Video Decoder HwDevice API

CMMFVideoDecodeHwDevice is the **Video Decoder HwDevice API**. To create a plugin, one should implement this interface and provide an ECom resource file to advertise the HwDevice to the **DevVideo** implementation. **CMMFVideoPostProcHwDevice** provides a **HwDevice** plugin for post processing. This API isn't explicitly discussed in this paper.

The hierarchy of classes that forms this API is shown below:

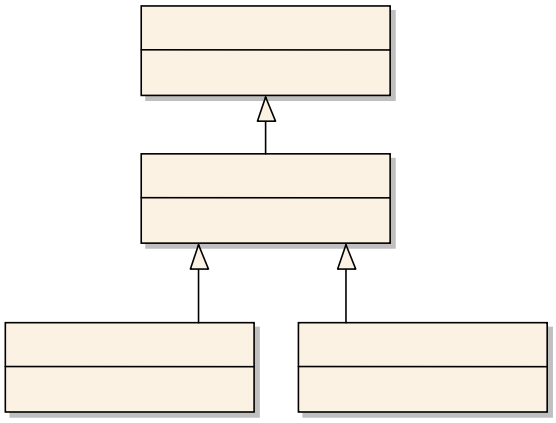


Figure 8 - Video decoder HwDevice API

A possible implementation of this API will look as the class below:

```
class COmxVideoDecodeHwDeviceAdapter :public CMMFVideoDecodeHwDevice
{
public:
static COmxVideoDecodeHwDeviceAdapter* NewL();
~COmxVideoDecodeHwDeviceAdapter();
// from CMMFVideoHwDevice
TAny* CustomInterface(TUid aInterface);
// from CMMFVideoPlayHwDevice
CPostProcessorInfo* PostProcessorInfoLC();
void GetOutputFormatListL(RArray<TUncompressedVideoFormat>& aFormats);
void SetOutputFormatL(const TUncompressedVideoFormat &aFormat);
void SetPostProcessTypesL(TUint32 aPostProcCombination);
void SetInputCropOptionsL(const TRect& aRect);
void SetYuvToRgbOptionsL(const TYuvToRgbOptions& aOptions, const TYuvFormat& aYuvFormat,
TRgbFormat aRgbFormat);
void SetYuvToRgbOptionsL(const TYuvToRgbOptions& aOptions);
void SetRotateOptionsL(TRotationType aRotationType);
void SetScaleOptionsL(const TSize& aTargetSize, TBool aAntiAliasFiltering);
void SetOutputCropOptionsL(const TRect& aRect);
void SetPostProcSpecificOptionsL(const TDesC8& aOptions);
void SetClockSource(MMMFClockSource* aClock);
void SetVideoDestScreenL(TBool aScreen);
void Initialize();
void StartDirectScreenAccessL(const TRect& aVideoRect, CFbsScreenDevice& aScreenDevice, const
TRegion& aClipRegion);
void SetScreenClipRegion(const TRegion& aRegion);
void SetPauseOnClipFail(TBool aPause);
void AbortDirectScreenAccess();
TBool IsPlaying();
void Redraw();
void Start();
```

```
void Stop();
void Pause();
void Resume();
void SetPosition(const TTimeIntervalMicroSeconds& aPlaybackPosition);
void FreezePicture(const TTimeIntervalMicroSeconds& aTimestamp);
void ReleaseFreeze(const TTimeIntervalMicroSeconds& aTimestamp);
TTimeIntervalMicroSeconds PlaybackPosition();
TUint PictureBufferBytes();
void GetPictureCounters(CMMFDevVideoPlay::TPictureCounters& aCounters);
void SetComplexityLevel(TUint aLevel);
TUint NumComplexityLevels();
void GetComplexityLevelInfo(TUint aLevel, CMMFDevVideoPlay::TComplexityLevelInfo& aInfo);
void ReturnPicture(TVideoPicture* aPicture);
TBool GetSnapshotL(TPictureData& aPictureData, const TUncompressedVideoFormat& aFormat);
void GetTimedSnapshotL(TPictureData* aPictureData, const TUncompressedVideoFormat& aFormat, const
TTimeIntervalMicroSeconds& aPresentationTimestamp);
void GetTimedSnapshotL(TPictureData* aPictureData, const TUncompressedVideoFormat& aFormat, const
TPictureId& aPictureId);
void CancelTimedSnapshot();
void GetSupportedSnapshotFormatsL(RArray<TUncompressedVideoFormat>& aFormats);
void InputEnd();
void CommitL();
void Revert();
// from CMMFVideoDecodeHwDevice
CVideoDecoderInfo* VideoDecoderInfoLC();
TVideoPictureHeader* GetHeaderInformationL(TVideoDataUnitType aDataUnitType,
TVideoDataUnitEncapsulation aEncapsulation, TVideoInputBuffer* aDataUnit);
void ReturnHeader(TVideoPictureHeader* aHeader);
void SetInputFormatL(const CCompressedVideoFormat& aFormat, TVideoDataUnitType aDataUnitType,
TVideoDataUnitEncapsulation aEncapsulation, TBool aDataInOrder);
void SynchronizeDecoding(TBool aSynchronize);
void SetBufferOptionsL(const CMMFDevVideoPlay::TBufferOptions& aOptions);
void GetBufferOptions(CMMFDevVideoPlay::TBufferOptions& aOptions);
void SetHrdVbvSpec(THrdVbvSpecification aHrdVbvSpec, const TDesC8& aHrdVbvParams);
void SetOutputDevice(CMMFVideoPostProcHwDevice* aDevice);
TTimeIntervalMicroSeconds DecodingPosition();
TUint PreDecoderBufferBytes();
void GetBitstreamCounters(CMMFDevVideoPlay::TBitstreamCounters& aCounters);
TUint NumFreeBuffers();
TVideoInputBuffer* GetBufferL(TUint aBufferSize);
void WriteCodedDataL(TVideoInputBuffer* aBuffer);
void SetProxy(MMMFDevVideoPlayProxy& aProxy);
void ConfigureDecoderL(const TVideoPictureHeader& aVideoPictureHeader);
private:
       OMX_HANDLETYPE iOmxHandle;
};
```

Table 2 Mapping of the Video Decoder HwDevice API calls to their OpenMAX IL equivalents

CMMFVideoDecodeHwDevice interface implementation	**OMX Methods**
NewL()	OMX_GetHandle();
SetOutputFormatL() SetPostProcessTypesL(); SetInputCropOptionsL(); SetYuvToRgbOptionsL(); SetYuvToRgbOptionsL(); SetRotateOptionsL(); SetScaleOptionsL(); SetOutputCropOptionsL(); SetPostProcSpecificOptionsL() SetBufferOptionsL() GetBufferOptions() GetHeaderInformationL() ConfigureDecoder() SetInputFormat() NumComplexityLevels() GetComplexityLevelInfo() SetComplexityLevel()	OMX_SetParameter()/OMX_SetConfig; OMX_GetParameter()/OMX_GetConfig;
Start();	OMX_SendCommand(OMX_StateExecuting)
Pause();	OMX_SendCommand(OMX_StatePaused)
Stop()	OMX_SendCommand(OMX_StateIdle)
Resume()	OMX_SendCommand(OMX_StateExecuting)
Intialise()	OMX_SendCommand(OMX_StateIdle)
WriteCodedDataL(…) GetBufferL() InputEnd()	Decoding buffer management OMX_EmptyThisBuffer() OMX_FillThisBuffer() EmptyBufferDone() FillBufferDone()
SetPosition(…) PlaybackPosition() DecodingPosition()	OMX_SetConfig(OMX_IndexConfigTimePosition) OMX_GetConfig(OMX_IndexConfigTimePosition)
FreezePicture() ReleaseFreeze()	OMX_SendCommand(OMX_StatePaused) OMX_SendCommand(OMX_StateExecuting)
GetOutputFormatList(…) SetClockSource(…) SetVideoDestScreenL(…) StartDirectScreenAccessL(…) SetScreenClipRegion(…) SetPauseOnClipFail(…) AbortDirectScreenAccess(…) IsPlaying() Redraw()	Internal Hw Device – not directly applicable

SetOutputDevice() SetProxy() PictureBufferBytes() GetPictureCounters() GetSnapShotL() GetTimedSnapshotL() CancelTimedSnapshot() GetSupportedSnapshotFormats() CommitL() Revert() VideoDecoderInfoLC() SynchronizeDecoding() SetHrdVrvSpec() PreDecoderBufferBytes() GetBitstreamCounters() NumFreeBuffers() ReturnHeader()	

4.3 Video Encoder HwDevice API

CMMFVideoEncodeHwDevice is the **Video Encoder HwDevice API**. To create a plugin, one should implement this interface and provide an ECom resource file to advertise the HwDevice to the **DevVideo** implementation. **CMMFVideoPreProcHwDevice** provides an **HwDevice** for preprocessing. This API isn't explicitly discussed in this paper.

The hierarchy of classes that forms this API is shown below:

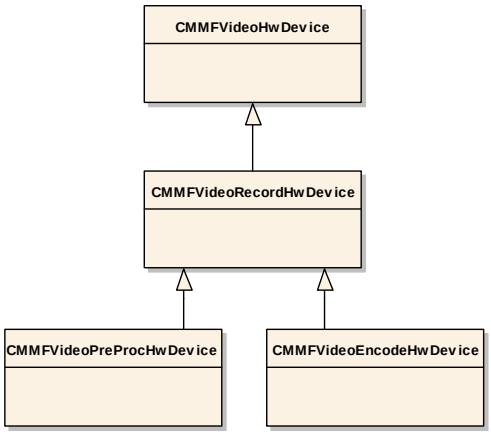


Figure 9 – CMMFVideoEncodeHwDevice API

```
class COmxVideoEncodeHwDeviceAdapter : public CMMFVideoEncodeHwDevice
{
```

```
public:
static COmxVideoEncodeHwDeviceAdapter* NewL()
// from CMMFVideoHwDevice
TAny* CustomInterface(TUid aInterface);
// from CMMFVideoRecordHwDevice
CPreProcessorInfo* PreProcessorInfoLC();
void SetInputFormatL(const TUncompressedVideoFormat& aFormat, const TSize& aPictureSize);
void SetSourceCameraL(TInt aCameraHandle, TReal aPictureRate);
void SetSourceMemoryL(TReal aMaxPictureRate, TBool aConstantPictureRate, TBool aProcessRealtime);
void SetClockSource(MMMFClockSource* aClock);
void SetPreProcessTypesL(TUint32 aPreProcessTypes);
void SetRgbToYuvOptionsL(TRgbRange aRange, const TYuvFormat& aOutputFormat);
void SetYuvToYuvOptionsL(const TYuvFormat& aInputFormat, const TYuvFormat& aOutputFormat);
void SetRotateOptionsL(TRotationType aRotationType);
void SetScaleOptionsL(const TSize& aTargetSize, TBool aAntiAliasFiltering);
void SetInputCropOptionsL(const TRect& aRect);
void SetOutputCropOptionsL(const TRect& aRect);
void SetOutputPadOptionsL(const TSize& aOutputSize, const TPoint& aPicturePos);
void SetColorEnhancementOptionsL(const TColorEnhancementOptions& aOptions);
void SetFrameStabilisationOptionsL(const TSize& aOutputSize,TBool aFrameStabilisation);
void SetCustomPreProcessOptionsL(const TDesC8& aOptions);
void Initialize();
void WritePictureL(TVideoPicture* aPicture);
void InputEnd();
void Start();
void Stop();
void Pause();
void Resume();
void Freeze();
void ReleaseFreeze();
TTimeIntervalMicroSeconds RecordingPosition();
void GetPictureCounters(CMMFDevVideoRecord::TPictureCounters& aCounters);
void GetFrameStabilisationOutput(TRect& aRect);
TUint NumComplexityLevels();
void SetComplexityLevel(TUint aLevel);
void CommitL();
void Revert();
// from CMMFVideoEncodeHwDevice
CVideoEncoderInfo* VideoEncoderInfoLC();
void SetOutputFormatL(const CCompressedVideoFormat& aFormat, TVideoDataUnitType aDataUnitType,
TVideoDataUnitEncapsulation aDataEncapsulation, TBool aSegmentationAllowed=EFalse);
void SetOutputRectL(const TRect& aRect);
void SetInputDevice(CMMFVideoPreProcHwDevice* aDevice);
void SetErrorsExpected(TBool aBitErrors, TBool aPacketLosses);
void SetMinRandomAccessRate(TReal aRate);
void SetNumBitrateLayersL(TUint aNumLayers);
```

```
void SetScalabilityLayerTypeL(TUint aLayer, TScalabilityType aScalabilityType);

void SetGlobalReferenceOptions(TUint aMaxReferencePictures, TUint aMaxPictureOrderDelay);

void SetLayerReferenceOptions(TUint aLayer, TUint aMaxReferencePictures, TUint
aMaxPictureOrderDelay);

void SetBufferOptionsL(const TEncoderBufferOptions& aOptions);

void SetCodingStandardSpecificOptionsL(const TDesC8& aOptions);

void SetImplementationSpecificEncoderOptionsL(const TDesC8& aOptions);

HBufC8* CodingStandardSpecificInitOutputLC();

HBufC8* ImplementationSpecificInitOutputLC();

void SetErrorProtectionLevelsL(TUint aNumLevels, TBool aSeparateBuffers);

void SetErrorProtectionLevelL(TUint aLevel, TUint aBitrate, TUint aStrength);

void SetChannelPacketLossRate(TUint aLevel, TReal aLossRate, TTimeIntervalMicroSeconds32
aLossBurstLength);

void SetChannelBitErrorRate(TUint aLevel, TReal aErrorRate, TReal aStdDeviation);

void SetSegmentTargetSize(TUint aLayer, TUint aSizeBytes, TUint aSizeMacroblocks);

void SetRateControlOptions(TUint aLayer, const TRateControlOptions& aOptions);

void SetInLayerScalabilityL(TUint aLayer, TUint aNumSteps, TInLayerScalabilityType
aScalabilityType,const TArray<TUint>& aBitrateShare, const TArray<TUint>& aPictureShare);

void SetLayerPromotionPointPeriod(TUint aLayer, TUint aPeriod);

HBufC8* CodingStandardSpecificSettingsOutputLC();

HBufC8* ImplementationSpecificSettingsOutputLC();

void SendSupplementalInfoL(const TDesC8& aData);

void SendSupplementalInfoL(const TDesC8& aData, const TTimeIntervalMicroSeconds& aTimestamp);

void CancelSupplementalInfo();

void GetOutputBufferStatus(TUint& aNumFreeBuffers, TUint& aTotalFreeBytes);

void ReturnBuffer(TVideoOutputBuffer* aBuffer);

void PictureLoss();

void PictureLoss(const TArray<TPictureId>& aPictures);

void SliceLoss(TUint aFirstMacroblock, TUint aNumMacroblocks, const TPictureId& aPicture);

void ReferencePictureSelection(const TDesC8& aSelectionData);

void SetProxy(MMMFDevVideoRecordProxy& aProxy);

private:

       OMX_HANDLETYPE iOmxHandle;

};
```

Table 3 Mapping of the Video Encoder Hw Device API calls to their OpenMAX IL equivalents

CMMFVideoEncodeHwDevice interface implementation	***OMX Methods***
NewL(…);	**OMX_GetHandle**(…);
SetInputFormatL(…); **SetSourceCameraL**(…); **SetSourceMemoryL**(…); **SetClockSource**(…); **SetPreProcessTypesL**(…); **SetRgbToYuvOptionsL**(…); **SetYuvToYuvOptionsL**(…); **SetRotateOptionsL**(…);	**OMX_SetParameter**(…); (only if applicable)

SetScaleOptionsL(…); **SetInputCropOptionsL**(…); **SetOutputCropOptionsL**(…); **SetOutputPadOptionsL**(…); **SetColorEnhancementOptionsL**(…); **SetFrameStabilisationOptionsL**(…); **SetOutputFormatL**(…); **SetErrorsExpected**(…); **SetMinRandomAccessRate**(…); **SetNumBitrateLayersL**(…); **SetScalabilityLayerTypeL**(…); **SetGlobalReferenceOptions**(…); **SetLayerReferenceOptions**(…); **SetBufferOptionsL**(…); **SetImplementationSpecificEncoderOptionsL**(…); **SetChannelPacketLossRate**(…); **SetChannelBitErrorRate**(…); **SetSegmentTargetSize**(…); **SetRateControlOptions**(…); **SetInLayerScalabilityL**(…); **SetLayerPromotionPointPeriod**(…); **SendSupplementalInfoL**(…); **SendSupplementalInfoL**(…);	
InputEnd(); **SetProxy**(…); **CommitL**(); **CustomInterface**(…); **PreProcessorInfoLC**(); **SetOutputRectL**(…); **SetInputDevice**(…); **SetErrorProtectionLevelsL**(…); **SetErrorProtectionLevelL**(…); **SetCustomPreProcessOptionsL**(…); **SetCodingStandardSpecificOptionsL**(…); **CommitL**(); **Revert**(); **GetPictureCounters**(…); **GetFrameStabilisationOutput**(…); **NumComplexityLevels**(); **SetComplexityLevel**();	**N/A**
Initialize();	**OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StateIdle);
Start();	**OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StateExecuting);
Pause();	**OMX_SendCommand**(iOmxHandle,

	OMX_CommandStateSet, OMX_StatePaused);
Stop();	**OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StateIdle);
Resume();	**OMX_SendCommand**(OMX_StateExecuting);
FreezePicture(); **ReleaseFreeze**();	**OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StatePaused) **OMX_SendCommand**(iOmxHandle, OMX_CommandStateSet, OMX_StateExecuting)
RecordingPosition();	**OMX_GetConfig**(OMX_IndexConfigTimePosition)
WritePictureL(…); **InputEnd**();	**OMX_EmptyThisBuffer**() **OMX_FillThisBuffer**() **EmptyBufferDone**() **FillBufferDone**()
NumComplexityLevels(…); **CodingStandardSpecificInitOutputLC**(); **ImplementationSpecificInitOutputLC**(); **CodingStandardSpecificSettingsOutputLC**(); **ImplementationSpecificSettingsOutputLC**(); **PictureLoss**(); **PictureLoss**(…); **SliceLoss**(…); **ReferencePictureSelection**(…);	**GetParameter**(); (only if applicable)

4.4 Call Sequences

The creation sequence for **Video Decoder/Encoder HwDevice** is very similar to the audio one as Figure 10shows.

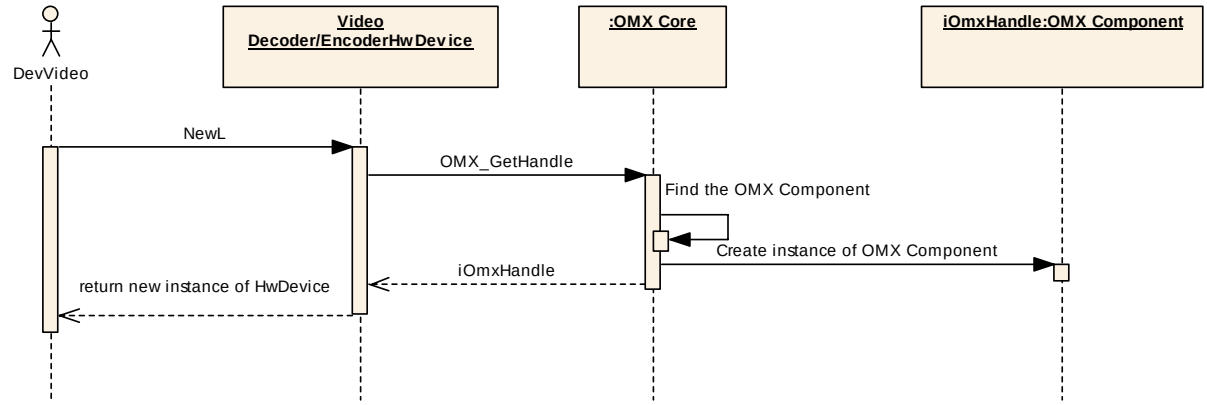


Figure 10 - Creation sequence for Video Decoder/Encoder HwDevice

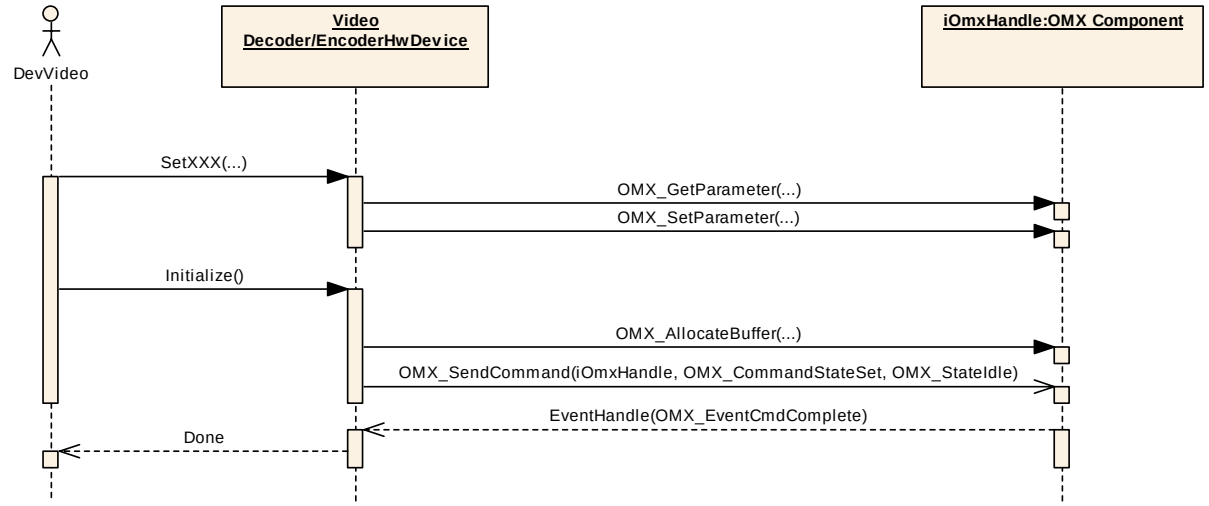


Figure 11 – Initialization of Video Decoder/Encoder HwDevice

In Figure 11 SetXX(…) stands for any method that configures the HwDevice.

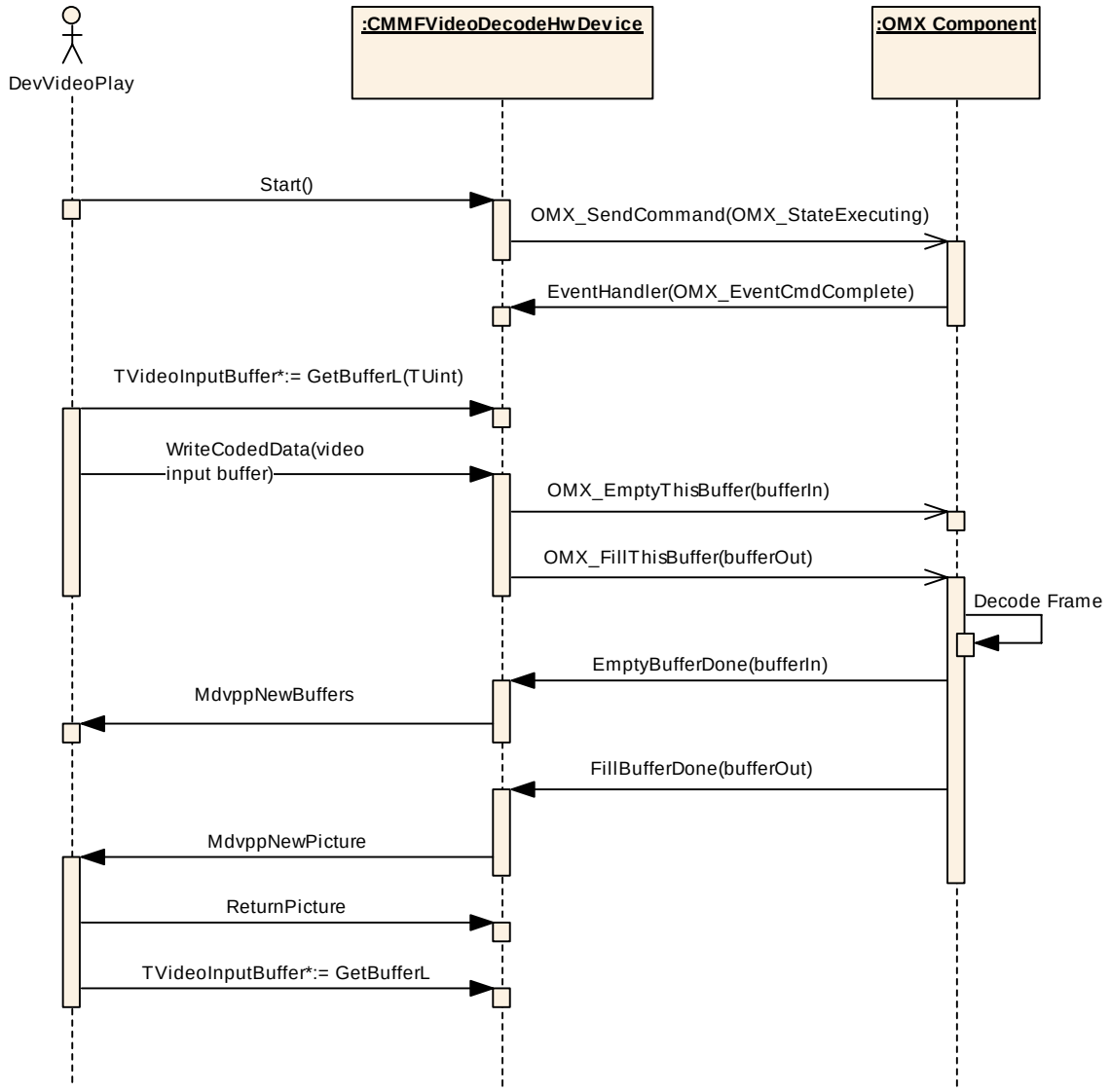


Figure 12 - Video Decoding Sequence

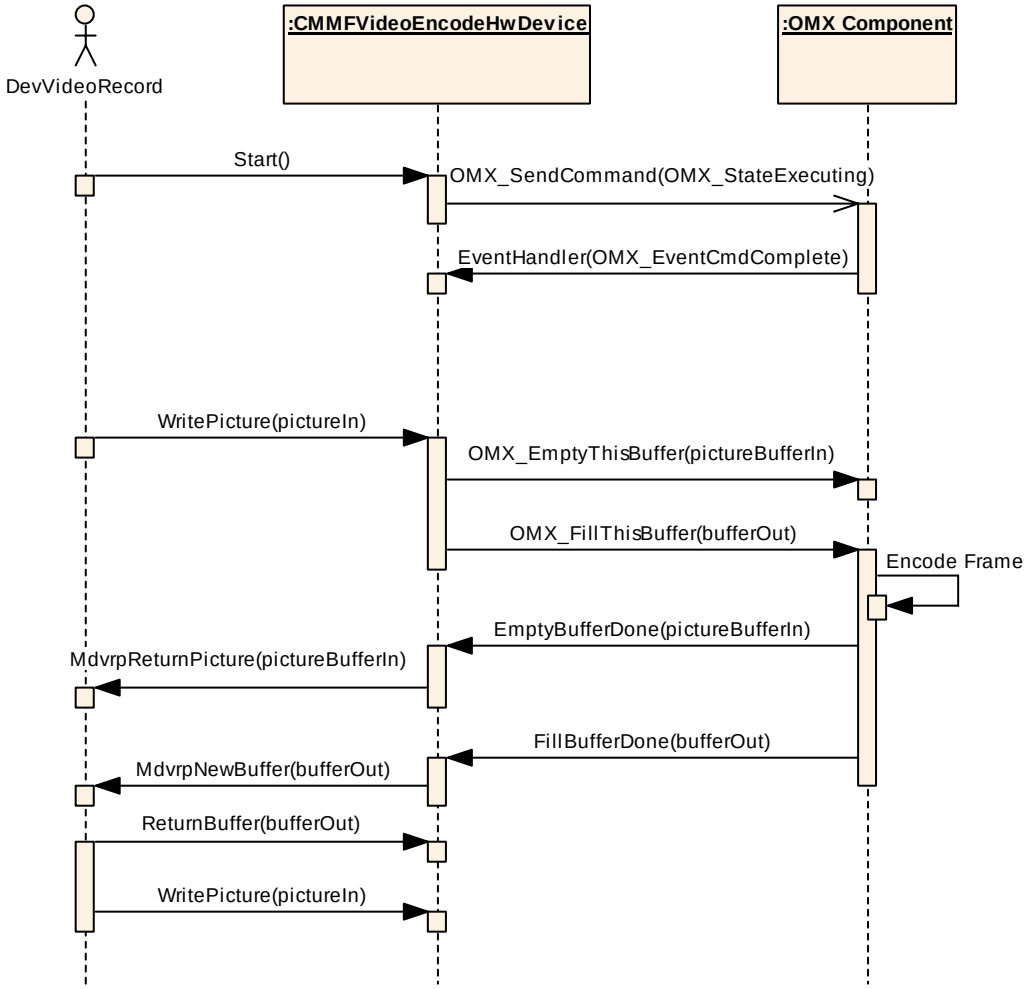


Figure 13 – Video encoding sequence

4.5 Video Buffers

4.5.1 Decoding

For decoding the **TVideoInputBuffer** structure is used for input buffers, and output picture frames are stored in **TVideoPicture**.

The input and output buffers are managed by the **HwDevice**, which can choose to allocate its own buffers, or request the IL component to allocate them. It will set the data pointers in the **TVideoInputBuffer** and the **TVideoPicture** structures.

The **TVideoInputBuffer** structure is shown in Figure 14. The iData field is a descriptor which points to the buffer. It holds the data pointer, length filled and maximum length of the buffer. The iOptions field represents which of the remaining fields are valid. It is the HwDevice's responsibility to copy the filled length and other valid data such as the time stamps to the OpenMAX buffer header.

```
class TVideoInputBuffer
        {
```

```
public:
	IMPORT_C TVideoInputBuffer();
public:
	enum TVideoBufferOptions
		{
		ESequenceNumber		= 0x00000001,
		EDecodingTimestamp	= 0x00000002,
		EPresentationTimestamp = 0x00000004
		};

	TPtr8 iData;
	TUint32 iOptions;
	TTimeIntervalMicroSeconds iDecodingTimestamp;
	TTimeIntervalMicroSeconds iPresentationTimestamp;

	TBool iPreRoll;
	TUint iSequenceNumber;
	TBool iError;
	TDblQueLink iLink;

	TAny* iUser;
	};
```

Figure 14 - TVideoInput Structure

4.5.2 Encoding

For encoding the **TVideoOutputBuffer** structure is used for output buffers, and input picture frames to be encoded are stored in **TVideoPicture**.

The **TVideoOutputBuffer** structure is shown in Figure 15. The **iData** field is a descriptor which points to the buffer. It is the **HwDevice's** responsibility to set the **iData** pointer from the OpenMAX buffer header and fill in any other information that it has from the IL component. A full discussion of the other fields is beyond the scope of this document.

```
class TVideoOutputBuffer
	{
public:
	TPtrC8 iData;
	TTimeIntervalMicroSeconds iCaptureTimestamp;
	TUint iCoverageStartPosition;
	TUint iCoverageEndPosition;
	TUint iOrderNumber;
	TUint iMinErrorProtectionLevel;
	TUint iMaxErrorProtectionLevel;

	TBool iRequiredSeveralPictures;
	TBool iRequiredThisPicture;
	TUint iLayer;
```

```
        TUint iSubSeqId;
        TUint iInLayerScalabilityStep;

        TUint iDataPartitionNumber;
        TBool iRandomAccessPoint;
        TPtrC8 iHrdVbvParams;
        TPtrC8 iCodingStandardSpecificData;
        TPtrC8 iImplementationSpecificData;
        TDblQueLink iLink;
        };
```

Figure 15 – TVideoOutputBuffer

4.5.3 Picture Buffers

The video picture structure **TVideoPicture** is shown in Figure 16. The data buffer is contained in the **iData** field which is of type **TPictureData**. The **iDataFormat** denotes the format of the data. It is necessary to use the **ERgbRawData**, or the **EYuvRawData** types, as one cannot use the data buffer of a CFbsBitmap due to the necessity of locking the font and bitmap server heap during the use of the buffer.

```
enum TImageDataFormat
        {
        /** Raw RGB picture data in a memory area.*/
        ERgbRawData           = 0x01000000,
        /** RGB picture data stored in a Symbian OS CFbsBitmap object. */
        ERgbFbsBitmap    = 0x02000000,
        /** Raw YUV picture data stored in a memory area. The data storage
        format depends on the YUV sampling pattern and data layout used.    */
        EYuvRawData           = 0x04000000
        };

class TPictureData
        {
public:
        TImageDataFormat iDataFormat;
        TSize iDataSize;
        union
                {
                TPtr8* iRawData;
                CFbsBitmap* iRgbBitmap;
                };
        };

class TVideoPicture
        {
public:
        enum TVideoPictureOptions
                {
                ETimestamp             = 0x00000001,
```

```
            ECropRect               = 0x00000002,
            EHeader                 = 0x00000004,
            EBitTargets             = 0x00000008,
            EReqInstantRefresh = 0x00000010,
            ESceneCut               = 0x00000020,
            EPictureEffect  = 0x00000040,
            EEffectParameters  = 0x00000080
            };
      TPictureData iData;
      TUint32 iOptions;
      TTimeIntervalMicroSeconds iTimestamp;
      TRect iCropRect;
      TVideoPictureHeader* iHeader;
      RArray<TUint>* iLayerBitRates;
      TPictureEffect iEffect;
      TUint iFadeParam;
      TAny* iUser;
      TDblQueLink iLink;
      };
```

Figure 16 - TVideoPicture structure

5 Tunneling and Platform Specific Notes

5.1 Audio

The audio **HwDevice** architecture limits the use of connected components to within the **HwDevice**. OpenMAX IL components may be tunneled as long as they exist in the same **HwDevice** component. This is due to the fact that there is a one to one relationship between HwDevices and DevSound sessions.

5.2 Video

In the video architecture, the decoding and postprocessor **HwDevices** for playback and the Preprocessor and Encoder **HwDevices** for recording are connected together.

If the pair of **HwDevices** are both implemented with IL components, they can use the **CustomInterface()** extension method to obtain a pointer to the IL component and thus allow the components to be directly tunneled.

6 Further Information

6.1 Glossary

Term	Definition

Term	Definition
Codec	Co)mpression/(Dec)ompression – coding algorithm used to compress/decompress digital multimedia samples for storage and transmission
Tunnelling	The direct connection of two components, so that they manage their own data transfer between components
ECom	Symbian plugin component architecture
CMMFHwDevice	Abstraction layer for codecs underneath the MDF DevSound layer

6.2 Acronym Definition Table

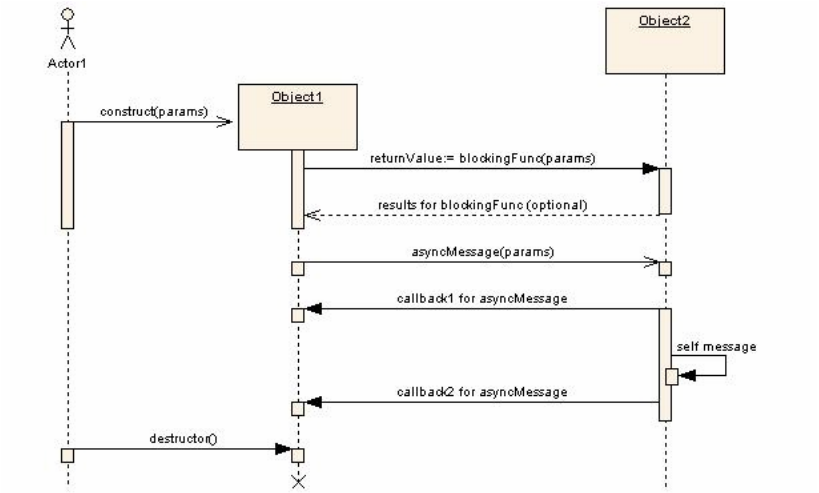
Term	Definition
MDF	Media Device Framework
MMF	Multimedia Framework
OpenMAX	The Standard for Media Library Portability
OMX	OpenMAX

6.3 UML notation for sequence diagrams

The UML sequence diagram notation used in this document is summarised as follows:

- A solid line with a solid arrow head is a blocking call - a return value may be specified.
- A dotted line with an open arrow head is a return and is optional.
- A solid line with an open arrow head is an asynchronous message.
- A solid line with an open arrow head sent to an object/class box indicates constructor call.

This notation is shown in the following diagram:



6.4 Document History

Date	Version	Status	Author	Description
2006-01-01	0.1	For Review	Kevin Butchart Viviana Dudau	First version to be reviewed
2006-04-24	1.0	Issued		First published version