OMX Core Integration Guide
OpenCore2.06, Rev 1
Mar 21, 2010

References

1*OpenMAX Integration Layer Application Programming Interface Specification.* Version 1.1.2, http://www.khronos.org/openmax/

2OpenMAX Call Sequences. http://android.git.kernel.org/?p=platform/external/opencore.git;a=tree;f=doc;hb=master

Table of Contents

List of Figures

1. Introduction

There are several ways to integrate a codec into the PV OpenCORE multimedia framework including as a compressed media I/O component, as a node, and as an OpenMAX component integrated into the OpenMAX codecs nodes that are part of the framework. Many codecs, especially those that include hardware acceleration, implement the OpenMAX IL interface making the OpenMAX interface the most straightforward method of integration in those cases.

One of the issues encountered in the integration process is the existence of multiple OMX cores. Even if all OMX components that come from different vendors are standard compliant, different vendors typically implement the same set of OpenMAX core methods (e.g. OMX_Init, OMX_Deinit, OMX_GetHandle, …). This poses a problem in terms of linking and executing the desired OMX core methods.

This document describes the details to help integration between the PV OpenCORE framework and OpenMAX IL 1.1 compliant components (encoders and decoders). While the OHA document "OpenMAX call sequences" describes the detailed interaction and call sequences between PV framework and OMX components, this document focuses on the integration of OMX core methods into the PV OpenCORE framework without causing conflicts with any other integrated OMX cores. Throughout this document the names PV framework and OpenCORE will be used interchangeably.

2. PV OMX test harness

2.1. General notes about the PV OMX test harness

Before integration of an OMX component begins, verification of the correct operation should be verified as a prerequisite. Using a set of conformance tests, the proper behavior of the component can be checked in isolation before introducing the additional complexity of the other components in the system. PV provides the PV OMX test harness for conformance testing of OMX components. .

The test harness **only allows one OMX core**, which may contain multiple OMX components, to be linked-in and tested at a time. In other words, the OMX core is tested in isolation at this step so there is no concern about co-existence of multiple OMX cores at this point.

2.2. PV OMX test harness location

The code for the PV OMX test harness are located in

```
.../codecs_v2/omx/omx_testapp/
```

for decoder components and

```
.../codecs_v2/omx/omx_testapp_enc
```

for encoder components.

2.3. Building the test harness with OMX plugins

Makefiles are provided to build the test harness including any OMX plugins. While the makefiles are setup to build the included test harness source code, they need to be modified to link in the OMX plugin library to be tested. These OMX plugin libraries must contain the standard OMX core methods such as OMX_init, OMX_Deinit, OMX_GetHandle, OMX_FreeHandle, etc.

The above mentioned makefiles are located in:

```
.../codecs_v2/omx/omx_testapp/build/makefile_omx_plugin
```

and in:

```
.../codecs_v2/omx/omx_testapp_enc/build/makefile_omx_plugin
```

2.4. Running test cases

The PV OMX test harness can be run from the command line. The documents provided in

```
.../codecs_v2/omx/omx_testapp/doc/...
.../codecs_v2/omx/omx_testapp_enc/doc/...
```

describe the PV OMX test harness command line arguments and test cases in elaborate detail. Once the tests in the test harness pass, the integration can proceed to the next step.

2.5. Handling partial compliance of OMX components

In some cases, an OMX component may not be fully compliant with the OpenMAX specification, but it is not feasible to correct the issues right away. For example, the component may not handle “OMX_UseBuffer” call, or it may not be capable of assembling partial input frames etc. In such cases, it may still be possible to integrate the component. However, the OMX component must inform the PV OpenCORE framework about its capabilities. This procedure is described in Section 3.2. of OHA document “OpenMAX call sequences”. In some cases, the capabilities flags can also be used to communicate preferences. For example, if a compliant OMX component can handle both “OMX_UseBuffer” and “OMX_AllocateBuffer” calls , but prefers to allocate its own buffers – it may still set the capability flags so as to force the framework to use the “OMX_AllocateBuffer” calls.

3. Building the OMX core library

3.1. OMX core methods

PV OpenCore provides an implementation of PV OMX core and PV OMX components. PV OpenCore framework behaves as OpenMax IL client when it communicates with OMX core and underlying OMX components. OMX core is responsible (among other things) for initializing, instantiating OMX components etc. Integration of OMX components into the OpenCore framework must be done by providing the OMX core to accompany the OMX components being integrated (all may be part of the same library). In other words, vendors should not attempt to integrate isolated OMX components (without the accompanying OMX core) into the existing PV OMX core – due to specialized APIs and communication between the PV OMX core and PV OMX components. Such APIs between PV OMX core and PV OMX components (for registering, instantiating PV OMX components) are internal, implementation specific, not prescribed by OMX specification and are subject to change at any time by PV. To ensure correct instantiation, registration and operation of its OMX components, a vendor should therefore implement and provide its own OMX core.

The OpenMAX Specification lists the following 9 methods that belong to the OMX core. All of the methods must be present in the library - although not all of them need to be fully supported. For example, if only OMX Base Profile is supported – the method OMX_SetupTunnel must be present in the OMX core library, but may simply return “OMX_ErrorNotImplemented”.

1)OMX_Init
2)OMX_Deinit
3)OMX_GetHandle
4)OMX_FreeHandle
5)OMX_ComponentNameEnum
6)OMX_GetComponentsOfRole
7)OMX_GetRolesOfComponent
8)OMX_SetupTunnel
9)OMX_GetContentPipe

3.2. Co-existence of multiple OMX cores

The main problem with the co-existence of multiple OMX cores – belonging to different vendors – is that all of them must implement the same set of OMX core methods. This can cause link issues if they were simply statically linked. One simple (but very impractical) method to resolve linking issues due to this problem would be to rename all OMX core methods in each vendor OMX core library to enable proper linking.

PV framework implements the method of dynamic loading/linking of multiple OMX cores. The OMX core methods are called through a thin wrapper layer (provided by PV). This wrapper layer contains the following method
1)OMX_MasterInit
2)OMX_MasterDeinit
3)OMX_MasterGetHandle

4)OMX_MasterFreeHandle
5)OMX_MasterComponentNameEnum
6)OMX_MasterGetComponentsOfRole
7)OMX_MasterGetRolesOfComponent
8)OMX_MasterSetupTunnel
9)OMX_MasterGetContentPipe

Dynamic loading is performed in OMX_MasterInit (code can be found in .../codecs_v2/omx/omx_mastercore/...) where all OMX cores and their components are discovered and registered. **Applications are required to call OMX_MasterInit and OMX_MasterDeinit to initialize and de-initialize all the OMX Cores.**

The above enables all OMX cores to keep the naming of OMX core methods, however – it imposes certain requirements on the OMX core library that is to be integrated. The requirements are described below.

3.3. Compiling the OMX core library

Because the PV OpenCORE framework is utilizing dynamic loading and linking to handle potentially multiple OMX cores simultaneously, the OMX core libraries must be built as shared objects, which typically means that the compiler flags must be set for position-independent code. It is fine to incorporate pre-built libraries (I.e., libraries built with an external build system) as long as it built in a compatible way as a shared object.

3.4. The OMX core library wrapper

The final shared object that contains the OMX core methods must also include the OMX core wrapper methods to enable correct dynamic loading of this library by the OpenCore framework.

There are currently two different build methods to include the PV OMX core wrapper into the final shared object library (.so):

3.4.1. Pre-built OMX core shared object library - adds OMX core wrapper separately

This method is used when the shared object library that contains OMX core methods is built separately (for example, included as a. pre-built binary), and it does not contain the OMX core wrapper. E.g. name such a library - “lib_prebuilt_omxcore_no_wrapper.so”

The OMX core wrapper must be added to this library separately to create and build a new shared library that contains both the “lib_prebuilt_omxcore_no_wrapper.so” and the OMX core wrapper interface. Appropriate makefiles must be created which will take the “lib_prebuilt_omxcore_no_wrapper.so” and add the OMX core wrapper to create this new library.

Call this new library e.g. “lib_omxcore_plus_wrapper.so”. This is the final OMX core library that can be dynamically loaded into OpenCore.

The code for the OMX core wrapper interface template for this case is provided in:

`.../codecs_v2/omx/omx_core_plugins/template/src`

The template code needs to be copied to an appropriate place in the directory structure of vendor's code that will enable the creation of “lib_omxcore_plus_wrapper.so”

Also, the template line:

#define OMX_CORE_LIBRARY "libOmxCore.so"

needs to be substituted with the following line that contains the name of the pre-built shared library that contains OMX core methods (but does not contain OMX core wrapper):

#define OMX_CORE_LIBRAY “ “lib_prebuilt_omxcore_no_wrapper.so”

As part of dynamic loading process, the OMX core wrapper for this case must open “lib_prebuilt omxcore_no_wrapper.so” library and link to the OMX core methods in this library explicitly (using dlopen & dlsym calls).

The provider of the final OMX core library must ensure that makefiles that create the final OMX library (“lib_omxcore_plus_wrapper.so”) include the OMX core library wrapper as well. The purpose of OMX core library wrapper is to provide the PV OpenCORE framework with common APIs to dynamically load and communicate with OMX core plugins.

3.4.2. OMX core wrapper is built simultaneously with OMX core methods

The second method (somewhat simpler than the first one) for creating a valid shared library that can be dynamically loaded by the OpenCore framework is the case when the code that contains OMX core methods and the OMX core wrapper interface can be built simultaneously.

In other words, the makefile that builds the OMX core methods needs to build the OMX core wrapper as well (at the same time) when creating the shared object.

In this case – the build process produces only one shared object library (e.g. call it “lib_omxcore_and_simultaneous_wrapper.so”) that can directly be used and can be dynamically loaded into OpenCore framework.

The code for the OMX core wrapper interface for this case is provided in:

```
.../codecs_v2/omx/omx_sharedlibrary/interface/src
```

This code needs to be copied to an appropriate place in the directory structure of vendor's code that will enable the creation of "lib_omxcore_and_simultaneous_wrapper.so"

3.5. The configuration parser API

In addition to standard OMX core methods, it is strongly recommended that the OMX core plugin library contains the configuration parser APIs. To optimize performance, PV framework needs to be able to determine whether a multimedia clip is valid and supported in order to do track selection and start playback. It's important that this process is efficient, and when multiple OMX cores are involved, it is best if this can be done prior to instantiating OMX components and codecs.

To facilitate this check, the OMX core wrapper and the PV OMX core library implement an API that allow the PV OpenCORE framework to obtain the information about audio/video track parameters (e.g. width, height, or sampling rate etc.) as well as the information about whether the OMX component and the underlying codec can handle the track. Based on this information, the PV OpenCORE framework makes a decision whether to proceed with playing the audio/video track or not.

NOTE: The implementation of the config parser API in an OMX core plugin library is optional, however – it is **strongly recommended**. If the config parser API is not provided in the OMX core plugin, the PV OpenCORE framework will fall back to the PV implementation of configuration parser API. However – since these APIs are custom made for the PV OMX components, this may mean that tracks unplayable by OMX core plugin library may be selected to play or it may also mean that playable tracks are rejected.

To implement the API, the method "OMXConfigParser" needs to be present in the OMX core plugin library that contains OMX core methods. The OMX core library wrapper will link to this method. If the wrapper fails to find the "OMXConfigParser" symbol, the wrapper will use the default pv config parser.

The API is defined as follows:

```
OMX_BOOL OMXConfigParser ( OMX_PTR aInputParameters,
                           OMX_PTR aOutputParameters);
```

where the input “aInputParameter” maps to (i.e. should be cast to a pointer to type):

```
typedef struct
{
      OMX_U8* inPtr;
      OMX_U32 inBytes;
      OMX_STRING cComponentRole;
      OMX_STRING cComponentName;
} OMXConfigParserInputs;
```

with

inPtr - pointer to codec configuration header
i**nBytes** – length of codec configuration header
cComponentRole – OMX component codec type (e.g., “video_decoder.mpeg4” or “audio_decoder.aac”)
cComponentName – OMX component name
Note: both component name and component role are needed as parameters because of the need by the OHA framework to be able to recognize the exact OMX core to which the component belongs and because a component may support multiple roles.

The return values and output parameters are:
OMXConfigParser returns OMX_FALSE if it cannot process the codec configuration header successfully or if the codec format/role is not supported at all.

OMXConfigParser returns OMX_TRUE if it can process the codec configuration header successfully. If the return value is OMX_TRUE, the OMXConfigParser is also expected to fill the “aOutputParameters” structure. The aOutputParameters structure in case of Audio codec maps to (i.e. should be cast to a pointer to type):

```
typedef struct
{
      OMX_U16 Channels;
      OMX_U16 BitsPerSample;
      OMX_U32 SamplesPerSec;
} AudioOMXConfigParserOutputs;
```

where

Channels – the detected number of channels in the audio track (e.g. 1 for mono, 2 for stereo,...)
BitsPerSample – bits per sample of audio (e.g. 16)
SamplesPerSec – Detected track sampling rate

In case of a video codec, the “aOutputParameters” structure maps to (i.e. should be cast to a pointer to type):

```
typedef struct
{
      OMX_U32 width;
```

```
        OMX_U32 height;
        OMX_U32 profile;
        OMX_U32 level;
    } VideoOMXConfigParserOutputs;
```

with

Width – the detected output video clip width
Height – the detected output video clip height
profile – if applicable – detected clip profile
level – if applicable – detected clip level

NOTE: Memory for OMXConfigParser input and output structures is allocated externally (i.e., there is no need to allocate the memory for the structures inside the OMXConfigParser method).

4. Dynamic loading and registration of OMX cores

An additional step is needed to register OMX core plugin libraries and enable the proper dynamic loading of these OMX cores into the PV OpenCORE framework.

The name of the OMX core library, which was built in the manner described above, needs to be recorded in the configuration file - together with the unique OsclUuid interface ID that corresponds to the PV OpenCORE framework API used for communicating with OMX core interfaces.

It is possible to place the information in an existing configuration file (e.g. pvplayer.cfg) or a new configuration file can be created that contains the required information (OsclUuid & library name). If a new configuration file is created, then the configuration file MUST have an extension ".cfg" and needs to be placed in "/system/etc" Android device directory. In either case, a new line is added to the *.cfg file containing:

```
(OMX Core API OsclUuid), "shared library name.so"
```

For example, to properly register and use the OMX core shared library (that contains vendor's OMX core and PV OMX core wrapper) named "libomx_core_vendorXYZ.so", the following line must be placed into the *.cfg file located in /system/etc directory.

```
(0xa054369c,0x22c5,0x412e,0x19,0x17,0x87,0x4c,0x1a,0x19,0xd4,0x5f),
"libomx_core_vendorXYZ.so"
```

NOTE: The OsclUuid provided is the unique API ID that identifies the OMX core interface and cannot be modified:

```
(0xa054369c,0x22c5,0x412e,0x19,0x17,0x87,0x4c,0x1a,0x19,0xd4,0x5f)
```

5. Details on data formats and OMX input buffers

5.1. General Notes about Data Formatting

This section provides additional information about the organization of codec data that is provided inside OMX buffers. Generally the OpenMAX IL spec [2] should provide sufficient details on the data formats, but there are some ambiguities and gaps in the specification that warrant the additional clarification.

5.1.1. Frame (or NAL) start codes

The term start codes refers to sequences in the bitstream that mark different boundaries such as frames or NALs in H.264. In general, start codes are NOT provided in most file formats and network protocols, and therefore the PV OpenCORE framework does NOT generally include start codes in the input buffers to OMX component. There are also no headers (e.g., headers that would provide the frame length, etc) artificially inserted into the bitstream within the input buffers.

5.1.2. OMX buffer fields

The “nFilledLen” OMX buffer field, “nTimestamp” OMX buffer field and OMX_BUFFERFLAG_ENDOFFRAME OMX buffer flag are used to communicate the size of data provided in the OMX buffer as well as frame (or NAL) boundaries. The OMX components should rely on this information to reconstruct the full frames/NALs if necessary. The nOffset is generally set to 0 by the PV OpenCORE IL client, but the OMX component should check the nOffset value in case it is non-zero.

5.1.3. Multiple frames in a single OMX input buffer

Exchanging a large number of relatively small buffers may negatively affect performance. Certain codec formats inherently pack data into packets that contain multiple frames (e.g., AMR IETF). In cases where input data frames are relatively small in size (e.g., AMR and most speech formats) – PV framework may pack multiple FULL frames into a single OMX input buffer. This is the case for most speech codecs. See sections below for details on which exact formats use packing of multiple frames of data into a single buffer. In such cases, OMX input buffers are still marked with OMX_BUFFERFLAG_ENDOFFRAME flag. This is illustrated in Figure 1 below.

Note that if packing of multiple frames into one OMX input buffer is used for a particular format – OMX input buffer contains one or more FULL frames of data of that format.
The “nTimestamp” OMX buffer field refers to the first frame of data in the buffer:

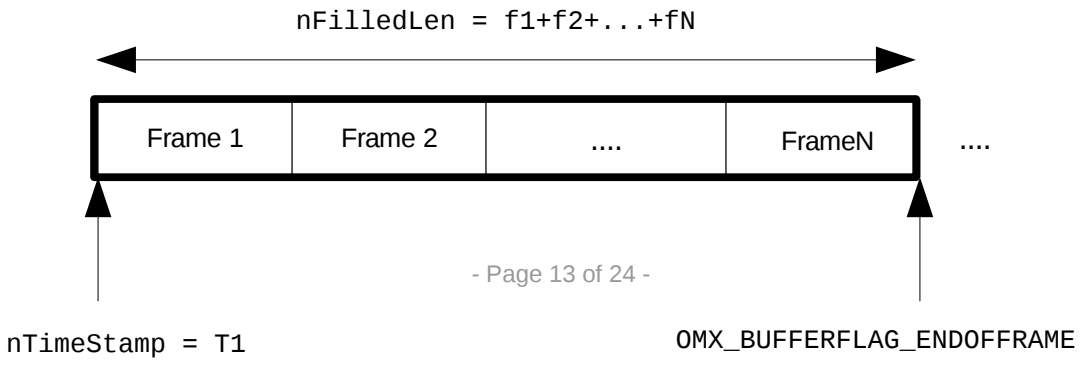

Figure 1: Packing multiple frames into one OMX buffer

5.1.4. Partial frames/NALs

In case of video formats, the PV OpenCORE framework may place either a full or a partial frame (or NAL) into an OMX input buffer. If a full frame or NAL is placed into the OMX input buffer, then the OMX_BUFFERFLAG_ENDOFFRAME flag is applied to that buffer. In case of partial frames/NALs, a frame/NAL may be split into multiple OMX input buffers and sent to the OMX component in pieces. Partial frames generally vary in size. In this case, only the buffer that contains the last piece of the frame/NAL is marked with the OMX_BUFFERFLAG_ENDOFFRAME flag. This is illustrated in and Figure 3 below.

In the case of video formats that allow partial frames, the PV OpenCORE framework NEVER places data that belongs to more than one frame/NAL in the same OMX input buffer which contains a partial frame/NAL. In streaming cases, it is also possible that data packets contain multiple FULL frames.

In other words, in the case of video formats that allow partial frames, each OMX input buffer can contain data that belongs to either:

–multiple FULL frames
–one full frame/NAL
–portion of a single frame/NAL

OMX_BUFFERFLAG_ENDOFFRAME flag (as well as nTimestamp field) enable the component to assemble partial frames.

In the case of formats that allow partial frames – each OMX input buffer that carries the data that belongs to the same frame shall have the same “nTimestamp” field. For example, if a frame is split into 3 OMX input buffers, all 3 input buffers will have the same timestamp field – and the last buffer will also have the OMX_BUFFERFLAG_ENDOFFRAME flag set.

NOTE: If necessary (i.e., if the OMX component is not capable of assembling partial frames), the PV OpenCORE framework can perform the assembly and provide a full frame to the OMX component. In order to enable this feature, it is necessary to use the “capability” flags. In other words, the OMX component must inform the PV OpenCORE framework about its capabilities. This procedure is described in Section 3.2. of the OpenMAX Call Sequences[2] document.

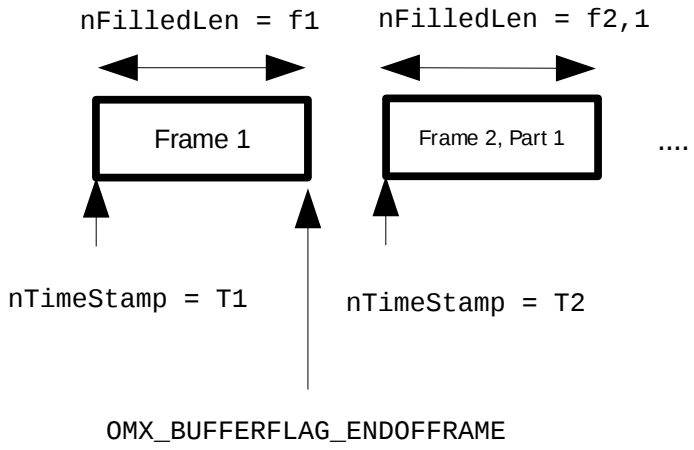

Figure 2: One full frame stored into one OMX buffer

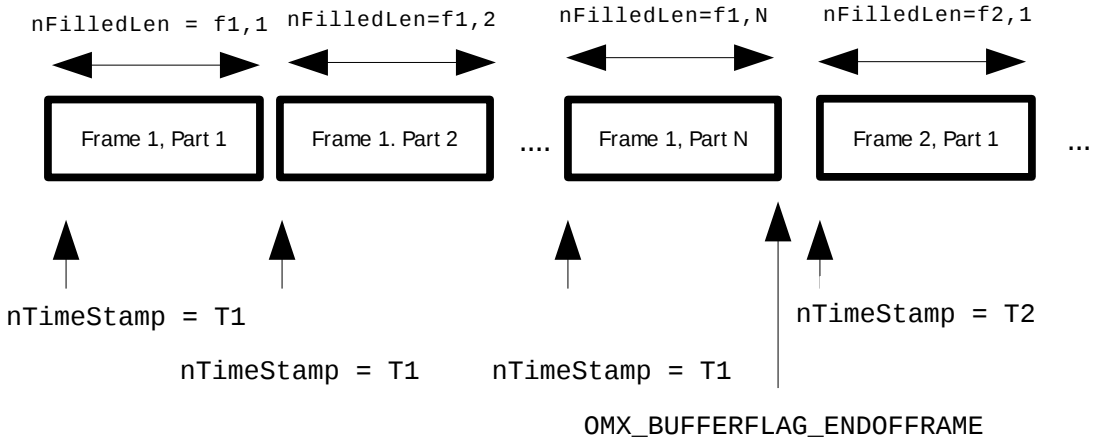

Figure 3: One frame split into N partial frames stored into N OMX buffers

5.1.5. Invalid data packing into OMX buffers

As mentioned above,

a) If PV framework uses packing of multiple frames per one OMX input buffer for a specific codec format, then PV framework will also ensure that only FULL frames will appear in such a buffer,

and that partial frames WILL NOT appear in any OMX buffers for that format. This situation is illustrated in Figure 4.

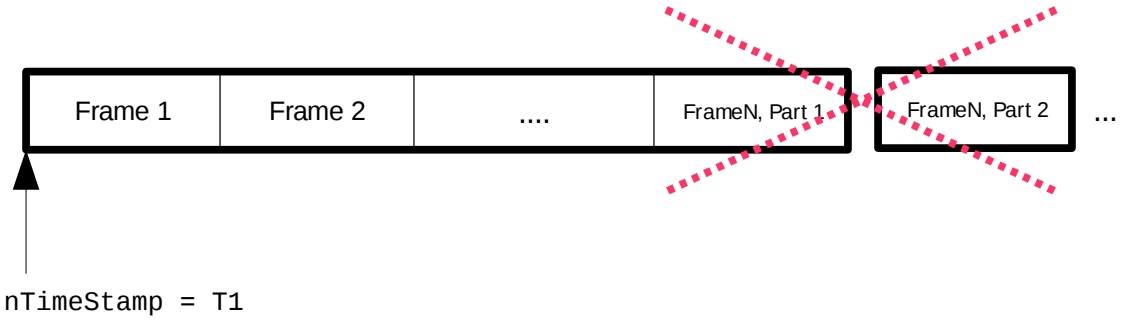

Figure 4: Invalid buffer content - If multiple frames are packed into a single OMX buffer, partial frames WILL NOT be stored in that buffer

b) If PV framework provides a frame/NAL split into partial frames/NALs for a specific codec format, then PV framework will also ensure that any data that belongs to multiple frames/NALs WILL NOT appear in the same OMX buffer. This is illustrated in Figure 5.

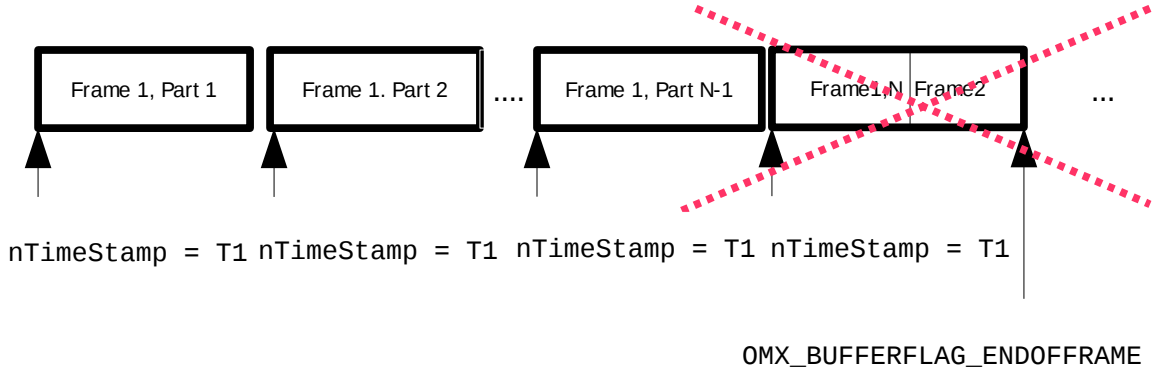

Figure 5: Invalid buffer content - If partial frames are used, data that belongs to more than one frame WILL NOT be stored in the same OMX buffer

5.1.6. Codec configuration data

If applicable, codec configuration data is sent in the first OMX input buffer. Codec configuration data buffer is marked with OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flags.

In case of H264 format, SPS and PPS NAL units are sent in separate OMX input buffers (both buffers are marked using OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flags).

5.2. AAC decoder formats

AAC decoder data can be stored in multiple formats. The data in the input OMX buffers is organized as follows:

5.2.1. ADIF AAC format

Codec Configuration Header: is sent in the first OMX input buffer. The first OMX input buffer contains only the codec configuration data and is marked with OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flag.

Data: ADIF AAC format does not provide frame boundaries. The ADIF data is placed into OMX input buffers sequentially and sent to the OMX component. The size of buffers may vary ("nFilledLen" buffer field informs the component about the quantity of data in the buffer).

Note: As a consequence of the lack of frame boundaries – except for the first OMX buffer (that contains the Codec Configuration Header) that provides the initial timestamp – all other OMX input buffer timestamps are invalid. However, the OMX component is expected to provide valid output buffer timestamps based on the initial input timestamp and the quantity of output PCM data that is produced.

5.2.2. ADTS AAC format

Codec Configuration Header is sent in the first OMX input buffer. The first OMX input buffer contains only the codec configuration data and is marked with OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flag.

Data: Each OMX input buffer contains one (or more than one) full ADTS frames. All OMX input buffers are marked with OMX_BUFFERFLAG_ENDOFFRAME. No partial frames are ever placed into OMX input buffer for this format. Since there can be multiple frames in a single OMX input buffer, the OMX component needs to keep consuming the data from the buffer until data is exhausted.

5.2.3. LATM AAC format

Codec Configuration Header is sent in the first OMX input buffer. The first OMX input buffer contains only the codec configuration data and is marked with OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flag.

Data: Each OMX input buffer contains one (or more than one) FULL LATM frames. All OMX input buffers are marked with OMX_BUFFERFLAG_ENDOFFRAME. No partial frames are ever placed into OMX input buffer for this format. Since there can be multiple frames in a single OMX input buffer, the OMX component needs to keep consuming the data from the buffer until data is exhausted.

5.2.4. MP4 audio AAC format

Codec Configuration Header is sent in the first OMX input buffer. The first OMX input buffer contains only the codec configuration data and is marked with OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flag.

Data: Each OMX input buffer contains one FULL MP4 Audio frame. All OMX input buffers are marked with OMX_BUFFERFLAG_ENDOFFRAME. No partial frames are ever placed into OMX input buffer for this format.

5.3. AMR decoder formats

The following applies to both Wide-band and Narrow-band versions of AMR.

Codec Configuration Header is NOT provided separately for this format. All the necessary configuration is provided through port parameters (e.g. IF2 vs. IETF)

Data: Each OMX input buffer contains one (or more than one) FULL AMR frames. All OMX input buffers are marked with OMX_BUFFERFLAG_ENDOFFRAME. No partial frames are ever placed into OMX input buffer for this format. AMR data formatting is based on the RFC 4867 document.

Note:
In case of RTP, the RTP packet is placed into the OMX input buffer (without the header). Based on RFC-4867 (RFC-3267), RTP AMR payload data is preceded by the table of contents which lists the frame types and indicates (implicitly) the number of frames present in the buffer.

File storage format and IF2: AMR data for IF2 or File Storage Format contains the frame type field followed by frame data. The frame type field determines the size of the frame that follows. Since there can be multiple frames in a single OMX input buffer, the OMX component needs to keep consuming the data from the buffer until data is exhausted.

5.4. MP3 decoder format

Codec Configuration Header is NOT provided separately for this format. Each frame of data carries its own header.

Data: Each OMX input buffer contains one (or more than one) FULL MP3 frames. All OMX input buffers are marked with OMX_BUFFERFLAG_ENDOFFRAME. No partial frames are ever placed into OMX input buffer for this format. Since there can be multiple frames in a single OMX input buffer, the OMX component needs to keep consuming the data from the buffer until data is exhausted.

5.5. WMA decoder format

Codec Configuration Header is sent in the first OMX input buffer. The first OMX input buffer contains only the codec configuration data and is marked with OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flag.

Data: Each OMX input buffer contains one (or more than one) FULL WMA frames. All OMX input buffers are marked with OMX_BUFFERFLAG_ENDOFFRAME. No partial frames are ever placed into OMX input buffer for this format. Since there can be multiple frames in a single OMX input buffer, the OMX component needs to keep consuming the data from the buffer until data is exhausted.

5.6. MPEG4 video decoder format

Codec Configuration Header (VOL header) s sent in the first OMX input buffer. The first OMX input buffer contains only the codec configuration data and is marked with OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flag.

Data: Each OMX input buffer contains either one or more full MPEG4 frames or a portion of a single frame. In case the buffer contains one or more full frames, OMX_BUFFERFLAG_ENDOFFRAME flag is applied to the OMX input buffer. In case the buffer contains a partial frame, OMX_BUFFERFLAG_ENDOFFRAME flag is applied to the input buffer only if it contains the last piece of that frame. If an OMX buffer contains a partial frame, then data belonging to more than one frame is NEVER placed in the same OMX input buffer .

Note: If necessary – i.e. if the OMX component is not capable of assembling partial frames, the PV OpenCORE framework can perform the assembly and provide a full frame to the OMX component. In order to enable this feature, it is necessary to use the “capability” flags. In other words, the OMX component must inform the PV OpenCORE framework about its capabilities. This procedure is described in Section 3.2. of OHA document “OpenMAX call sequences”. Frame assembly in the PV OpenCORE framework may incur a performance penalty.

5.7. H263 video decoder format

Codec Configuration Header is NOT provided separately for this format. Each frame of data carries its own header.

Data: Each OMX input buffer contains either one or more full H263 frames or a portion of a single frame. In case the buffer contains one or more full frames, OMX_BUFFERFLAG_ENDOFFRAME flag is applied to the OMX input buffer. In case the buffer contains a partial frame, OMX_BUFFERFLAG_ENDOFFRAME flag is applied to the input buffer only if the buffer contains the last piece of that frame. If an OMX buffer contains a partial frame, then data belonging to more than one frame is NEVER placed in the same OMX input buffer.

Note1: H263 is the only format where the very first OMX input buffer might not be marked with OMX_BUFFERFLAG_ENDOFFRAME.

Typically, for all other PV supported formats - the first OMX input buffer either contains the codec configuration header or is guaranteed to contain a full frame. In both of these cases – the very first OMX input buffer is always marked with OMX_BUFFERFLAG_ENDOFFRAME. This may not

be the case for H263 since the first OMX input buffer may contain the first piece of the first h263 frame.

Note2: If necessary – i.e. if the OMX component is not capable of assembling partial frames, the PV OpenCORE framework can perform the assembly and provide a full frame to the OMX component. In order to enable this feature, it is necessary to use the “capability” flags. In other words, the OMX component must inform the PV OpenCORE framework about its capabilities. This procedure is described in Section 3.2. of OHA document “OpenMAX call sequences”. Frame assembly in the PV OpenCORE framework may incur a performance penalty.

5.8. WMV decoder format

Codec Configuration Header is sent in the first OMX input buffer. The first OMX input buffer contains only the codec configuration data and is marked with OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flag.

Data: Each OMX input buffer contains either one or more full WMV frames or a portion of a single frame. In case the buffer contains a full frame, OMX_BUFFERFLAG_ENDOFFRAME flag is applied to the OMX input buffer. In case the buffer contains a partial frame, OMX_BUFFERFLAG_ENDOFFRAME flag is applied to the input buffer only if it contains the last piece of that frame. If an OMX buffer contains a partial frame, then data belonging to more than one frame is NEVER placed in the same OMX input buffer.

Note: If necessary – i.e. if the OMX component is not capable of assembling partial frames, the PV OpenCORE framework can perform the assembly and provide a full frame to the OMX component. In order to enable this feature, it is necessary to use the “capability” flags. In other words, the OMX component must inform the PV OpenCORE framework about its capabilities. This procedure is described in Section 3.2. of OHA document “OpenMAX call sequences”. Frame assembly in the PV OpenCORE framework may incur a performance penalty.

5.9. H264/AVC decoder format

Codec Configuration Header:
SPS and PPS NAL units are sent in the first OMX input buffers (the order in which SPS and PPS are sent is not guaranteed). SPS and PPS NALs are sent in separate buffers and these input buffers are marked with OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_CODECCONFIG flag.

5.9.1. AVC NAL Mode vs. AVC Frame mode

AVC data can be provided to the OMX component in two different modes depending on the setting of the capability flag “iOMXComponentUsesFullAVCFrames” (described in in Section 3.2. of OHA document “OpenMAX call sequences”). The default is NAL mode and in this mode, the OpenCore framework provides a single or a partial AVC NAL in an OMX input buffer at a time. In the Frame mode – OpenCore framework accumulates AVC NALs and provides the OMX component with one full AVC frame placed into an OMX input buffer. NAL boundaries are communicated to the OMX component using OMX_OTHER_EXTRADATA structures. If both

iOMXComponentUsesFullAVCFrames" and "iOMXComponentUsesNALStartCodes" capability flags are set to OMX_TRUE – then NAL boundaries inside the frame can be inferred by parsing the inserted NAL start codes. In such a case – OMX_OTHER_EXTRADATA structures are not used.

Data structure – NAL MODE: Each OMX input buffer contains either one full AVC NAL or a portion of a NAL. In case the buffer contains a full NAL, OMX_BUFFERFLAG_ENDOFFRAME flag is applied to the OMX input buffer. In case the buffer contains a partial NAL, OMX_BUFFERFLAG_ENDOFFRAME flag is applied to the input buffer only if it contains the last piece of that NAL. Data belonging to more than one NAL is NEVER placed in the same OMX input buffer for this format.

Data structure – Frame Mode: Each OMX input buffer contains one full AVC frame. In case where NAL start code insertion is requested by the OMX component, NAL boundaries can be inferred by the OMX component by parsing the OMX input buffer for NAL start codes. In case where NAL start code insertion is NOT requested by the OMX component (default), NAL boundaries (i.e. NAL lengths) are communicated to the OMX component using OMX_OTHER_EXTRADATA structure as defined in section 4.2.33 of the OpenMax IL spec version 1.1.2.
In Frame Mode, OMX_BUFFERFLAG_ENDOFFRAME flag is applied to each and every OMX input buffer. Each buffer in Frame mode SHALL have the OMX_BUFFERFLAG_EXTRADATA bit set in the nFlags field of the OMX_BUFFERHEADERTYPE structure.

At the end of the buffer that contains the AVC frame (i.e. after nFilledLen+nOffset bytes counting from the beginning of the buffer)– the following data is appended:

OMX_OTHER_EXTRADATATYPE extra;
OMX_OTHER_EXTRADATATYPE terminator;

where:

extra.eType = OMX_ExtraDataNALSizeArray;
extra.nSize = 20+4*(number of NALs in the frame); // 20 is the size of OMX_OTHER_EXTRADATATYPE structure + 4 bytes per NAL size
extra.nDataSize = 4 * (number of NALs in the frame)
extra.data[4*i] = size of the i-th NAL (data is declared as byte array – so offset is 4*i, since 4 bytes is assigned to signal the size of each NAL unit)

terminator.eType = OMX_ExtraDataNone;
terminator.nSize = 20;
terminator.nDataSize = 0;

Also, "OMX_ExtraDataNALSizeArray" is a custom value defined as:

#define OMX_ExtraDataNALSizeArray 0x7F123321

By reading and interpreting the OMX_OTHER_EXTRADATA structures – the OMX component

can determine the number of NAL units and the size of each NAL unit in the AVC frame – which is sufficient to determine NAL boundaries.

Figure 6 shows an example of an OMX buffer in AVC Frame Mode with 2 NALs and with the length values included in the extra data at the end of the buffer. The example is in a similar format to Figure 4-3 of [2]. The following is a summary of details for handling the NAL lengths:
1.Each buffer that includes NAL length data SHALL have the OMX_BUFFERFLAG_EXTRADATA bit set in the nFlags field of the OMX_BUFFERHEADERTYPE structure.
2.The length of each NAL SHALL be encoded as a single 4 byte unsigned integer (i.e., the OMX_U32 type as defined in the OpenMAX specification).
3.The data for the set of all NAL lengths contained in the buffer SHALL be encoded as an array of type OMX_U32 that is located at the end of the buffer.

4.Buffers containing complete frames SHALL have the OMX_BUFFERFLAG_ENDOFFRAME bit in the nFlags field of the OMX_BUFFERHEADERTYPE set.
5.A single buffer SHALL NOT contain data from multiple frames.

Note 1 – Applies to NAL Mode Only: H264/AVC format is NAL based and applies OMX_BUFFERFLAG_ENDOFFRAME to mark the NAL (and NOT frame) boundaries. It is possible to infer the frame boundary based on the “nTimestamp” buffer fields which contain the same timestamp for all NALs and portions of NALs that belong to the same frame.

Note 2 – Applies to both NAL and Frame mode: There is no NAL start codes in the bitstream, and by default, NAL start codes are not inserted by the PV OpenCORE framework. If necessary, it is possible for the PV OpenCORE framework to insert NAL start codes at the beginning of each NAL (with a performance penalty). In order to enable this feature, it is necessary to use the “capability” flags. In other words, the OMX component must inform the PV OpenCORE framework about its capabilities. This procedure is described in Section 3.2. of OHA document “OpenMAX call sequences”.

Note3 – Applies to both NAL and Frame mode: If necessary – i.e. if the OMX component is not capable of assembling partial NALs, the PV OpenCORE framework can perform the assembly and provide a full NAL to the OMX component. Also – if necessary, the OpenCore framework can perform the collection of NALs into an AVC frame. The two features can be used separately. In order to enable these features, it is necessary to use the “capability” flags. In other words, the OMX component must inform the PV OpenCORE framework about its capabilities. This procedure is described in Section 3.2. of OHA document “OpenMAX call sequences”. NAL and frame assembly in the PV OpenCORE framework may incur a performance penalty.

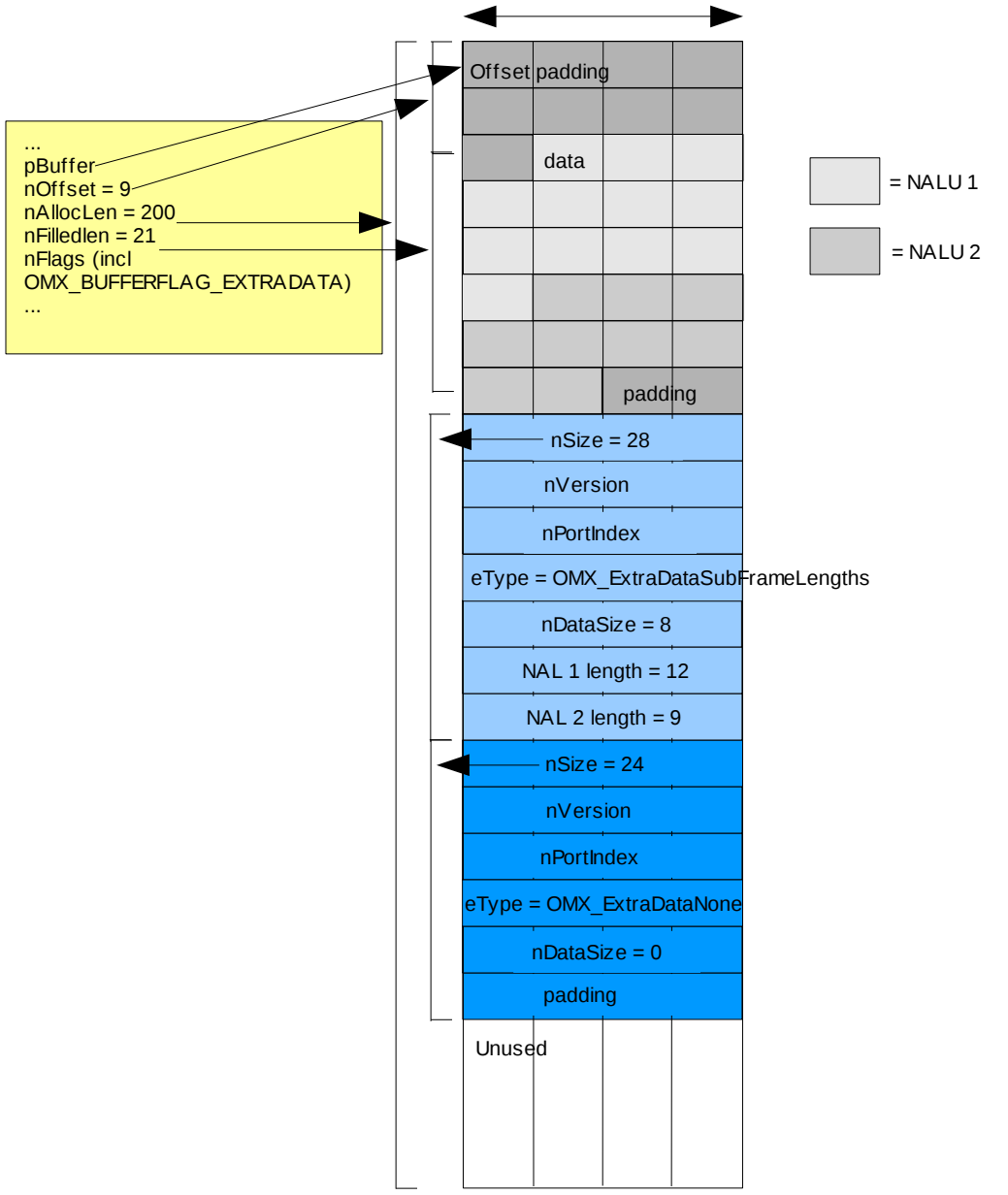

Figure 6: Example of NAL lengths in the buffer extra data.

5.10. YUV/RGB data format

In case of OMX video encoder components, raw video data is provided in either YUV or RGB format. The PV OpenCORE framework will provide one FULL frame of YUV or RGB data to OMX components.