Guide to Supplying Decoder Buffers from the MIO Component
OpenCORE 2.02, ver 1
Mar 13, 2009

Table of Contents

List of Figures

References

1 *Media I/O Developer's Guide.* OpenCORE 2.02, rev. 1. *http://android.git.kernel.org/?p=platform/external/opencore.git;a=summary*
2 *OpenMAX Integration Layer Application Programming Interface Specification.* Version 1.1.2, http://www.khronos.org/openmax/

1. Introduction

In some situations it may be more efficient for the rendering component to allocate and supply the decoder's output buffers, which are the inputs to the renderer. There may be constraints for the input buffers to the rendering components such as the buffers must be physically contiguous, reside in specific memory regions, etc. In order to avoid copying, which can be quite expensive for video, the decoder must place its output directly into buffers that meet the constraints for the renderer input buffers.

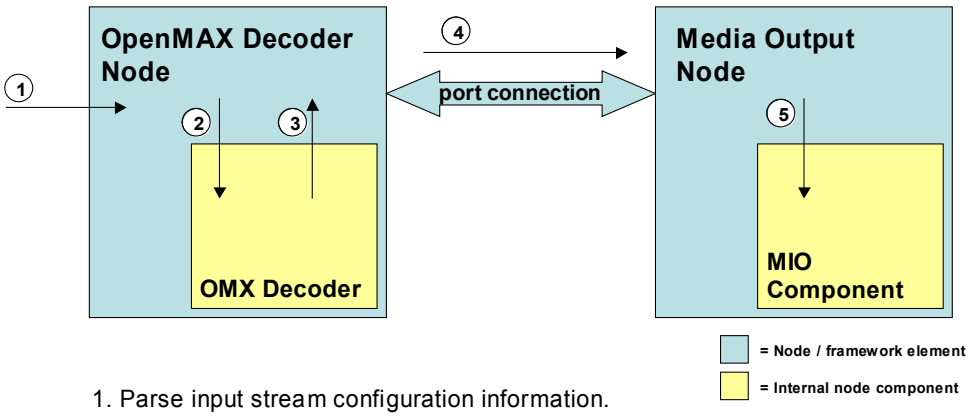
This document details the methods for providing decoder output buffers from the media I/O component. Although the methods described are primarily useful for video, they are not limited to video. Familiarity with media I/O components is assumed throughout the document. See the Media I/O Developer's Guide[1] for those details.

2. Overview of Graph Initialization

During the initialization of the multimedia graph for playback, nodes implementing data sources, decoding, and rendering are connected, parameters are configured, and buffers are allocated. The buffer allocation is established as a part of the graph setup phase between each connected pair of nodes. The media data in the basic playback graph tends to flow in one direction from source to sink, and the typical case is for the upstream node to allocate and supply buffers to the downstream node However, the scenario of interest here is one where the downstream sink node allocates and supplies buffers to the upstream decoder node. The methods in this document only concern the decoder and media output nodes, so the description and diagrams will focus only on that portion of the multimedia graph for clarity.

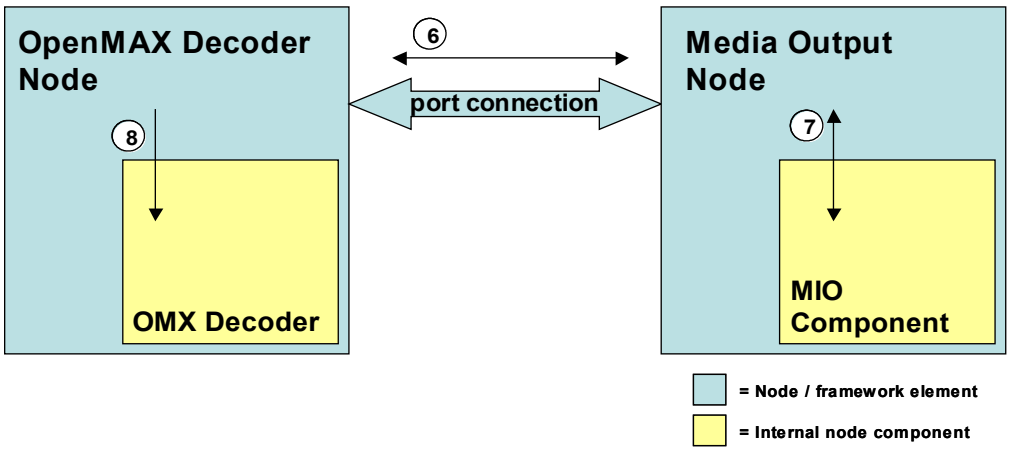
Figure X and Figures Y illustrate a high-level overview of the sequence of messages involved in the initialization of the decoder and media I/O nodes for video. The figures show the communication between the nodes as well as the communication with each node's internal components. Although the example mentions specific parameters for video, the process applies generally with main difference being the parameters that are passed along with the desired number of buffers and buffer size. For audio, those parameters would be number of channels, sample rate, and bits per sample.

After passing the basic configuration information (e.g., format, width, height, etc for video) including the desired number of buffers and buffer size, the decoder node will query the media output node for an allocator object. The MIO component may optionally provide an allocator at this point. If an allocator is provided, then the decoder will use it for allocating buffers. Otherwise, the decoder node handles the buffer allocation internally. The decoder node will use the standard OpenMAX[2] OMX_UseBuffer calls to notify the OpenMAX component about these buffers, so no special handling is required within the OpenMAX component.



1. Parse input stream configuration information.
2. Use OMX SetParameter calls to set Width x Height information.
3. Use OMX GetParameter to query desired number of buffers and buffer size.
4. Pass Width x Height, number of buffers, buffer size over the port interface.
5. Pass Width x Height, number of buffers, buffersize over SetParameterSync call

Figure 1: High-level Initialization Sequence (Part 1)



6. Use GetParameterSync call over the port to query for buffer allocator.
7. Use GetParameterSync calls to query for buffer allocator.
8. After allocating buffers and wrapping inside memory pool, use UseBuffer calls to pass buffers to OMX decoder.
9. At this point, buffer passing proceeds as usual using existing APIs.

Figure 2: High-level Initialization Sequence (Part 2)

3. Sequence Diagrams

This section contains more detailed sequence diagrams showing the call flow between the different components including the allocator object. Note that the allocator is only used initially to obtain the buffers. During steady-state processing the buffer usage is tracked with other data container objects and the buffers are recycled once they are done being rendered.

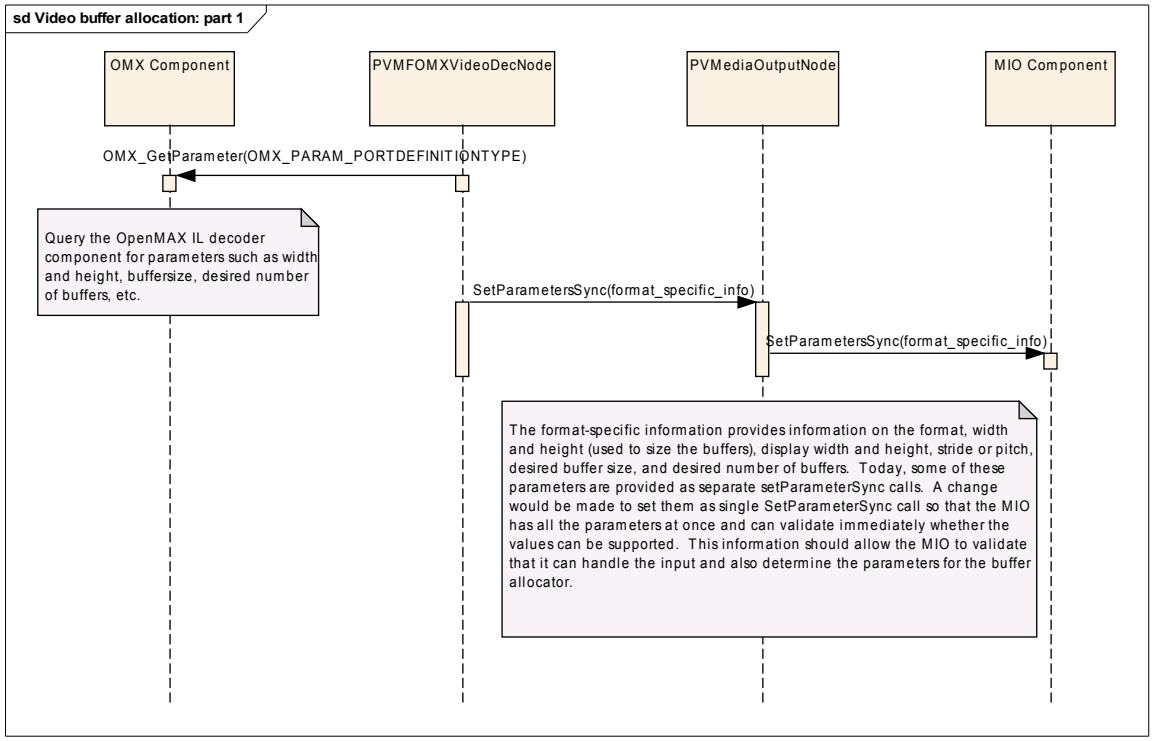


Figure 3: Video Buffer Allocation (Part 1)

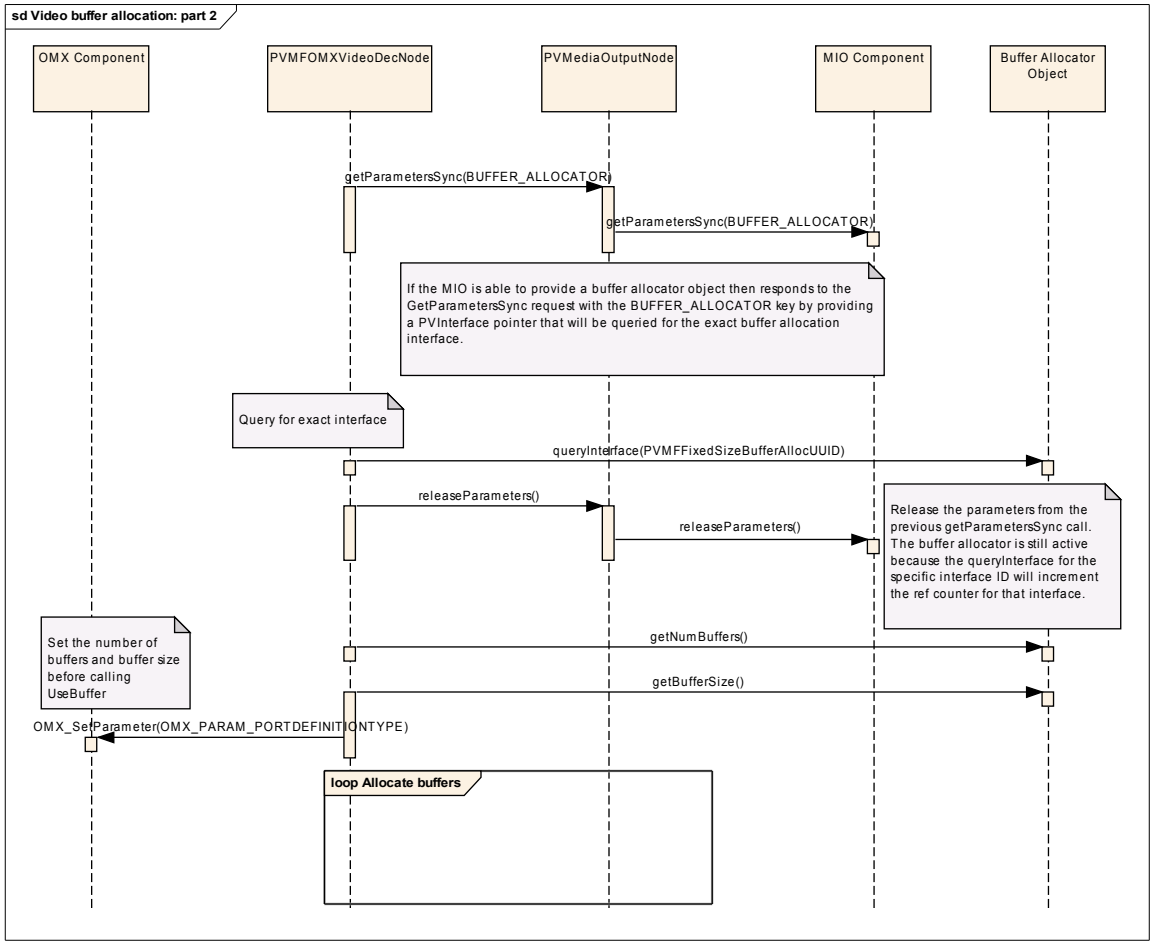


Figure 4: Video Buffer Allocation (Part 2)

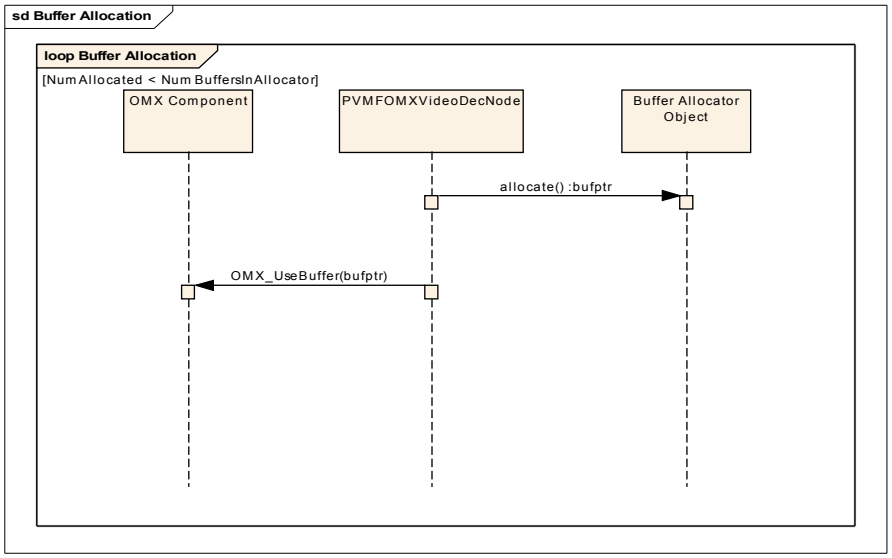


Figure 5: Buffer Allocation

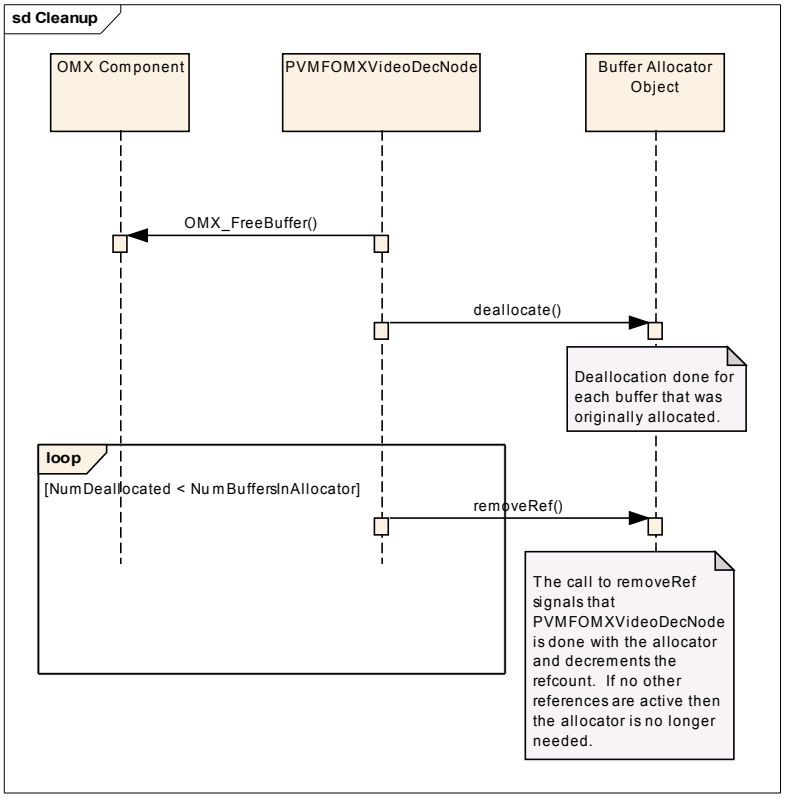


Figure 6: Cleanup

4. Buffer Allocator

The buffer allocator has a very simple interface that provides a way to allocate and release buffers of a fixed size. The interface also provides methods to get the fixed buffer size and the number of buffers that the allocator can provide. The interface definition is provided below for convenience, but the definition in the header file should be referenced for the most up-to-date version.

```
/**
 * This interface is used to allocate a set of fixed-size buffers.
 */
class PVMFFixedSizeBufferAlloc
{
    public:

        virtual ~PVMFFixedSizeBufferAlloc() {};

        /**
         * This method allocates a fixed-size buffer as long as there are
         * buffers remaining.  Once the maximum number of buffers have been
         * allocated, further requests will fail.
         *
         * @returns a ptr to a fixed-size buffer
         * or NULL if there is an error.
         */
        virtual OsclAny* allocate() = 0;

        /**
         * This method deallocates a buffer ptr that was previously
         * allocated through the allocate method.
         *
         * @param ptr is a ptr to the previously allocated buffer to release.
         */
        virtual void deallocate(OsclAny* ptr) = 0;

        /**
         * This method returns the size of the buffers that
         * will be allocated.
         *
         * @returns the fixed size used for all buffers.
         */
        virtual uint32 getBufferSize() = 0;

        /**
         * This method returns the maximum number of buffers
         * available for allocation
         *
         * @returns the max number of buffers available for allocation.
         */
        virtual uint32 getNumBuffers() = 0;
};
```