Media I/O Developer's Guide
OpenCORE 2.02, rev. 1
March 13, 2009

Table of Contents

List of Figures

References

1*OpenMAX Integration Layer Application Programming Interface Specification.* Version 1.1.2, http://www.khronos.org/openmax/
2*PVPlayer SDK Developer's Guide.* OHA 1.0, rev. 2. http://android.git.kernel.org/?p=platform/external/opencore.git;a=summary
3*Guide to Supplying Decoder Buffers from the MIO Component*. OpenCORE 2.02, rev 1. http://android.git.kernel.org/?p=platform/external/opencore.git;a=summary

1 Introduction

In the PV multimedia framework (PVMF) architecture, the Media I/O (MIO) component is a data sink or source at either the beginning or end of the datapath for media data. It is responsible for rendering media data in the case of playback or capturing media data in the case of authoring.. As access to media rendering functionalities are different for every platform that PV software works on, the MIO component also serves as a glue layer between PV modules and the media decoder and renderer hardware. MIO component can also function as a pass through module for media rendering to be done by the application. MIO components can contain logic to support application level features that involves renderer control.

In order to act as an adapter between PV modules and underlying hardware, an MIO component has several responsibilities.

- Control the underlying hardware based on commands from other PV modules.
- Exchange capabilities information of the underlying hardware with other PV modules.
- Provide media data to the underlying hardware for the data to be rendered to user at the appropriate time.

The purpose of this document is to provide detailed but platform agnostic guidance for MIO component developers. Developers are expected to derive from this platform agnostic guidance to design and implement MIO components customized for a specific target platform and hardware. Developers using this document are expected to have reasonable prior knowledge about overall architecture of PVPlayer integration, and PV core technologies such as OSCL and PVMI framework. For further information about PV core technologies, please consult the documents listed in the Reference section of this document.

2 Overview

2.1 Controls

The PV multimedia framework defines the PvmiMIOControl interface to allow other modules to control the MIO based on control commands from the end user. This section explains how the MIO is controlled based on application level commands, and how MIO in turn controls the hardware.

The PvmiMIOControl interface has a number of asynchronous commands, and therefore the MIO component is expected to maintain a command queue of incoming control commands, and process them in a FIFO order. The only exception to the FIFO order is handling of CancelCommands and CancelAllCommands which will be discussed in sections 22and 27. When the processing of asynchronous control commands is completed, the MIO component should call the RequestCompleted callback defined in PvmiMIOObserver to complete the control command.

The sequence diagram in Figure 1 below illustrates the general sequence of handling asynchronous PvmiMIOControl commands.

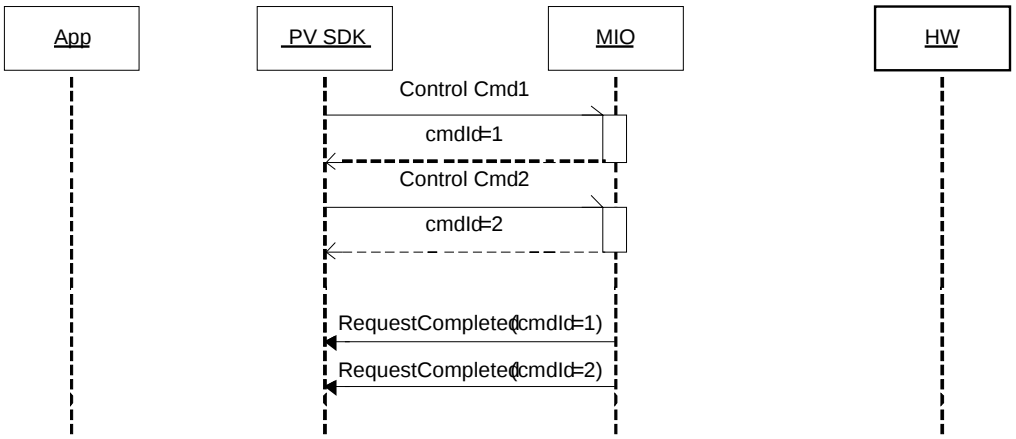

Figure 1: Sequence diagram showing asynchronous MIO commands.

2.2 PVMI Interfaces used by MIO Components

PVMF defines several interfaces for an MIO component to implement such that it can provide the necessary functionality to adapt between various PV modules and the underlying hardware.

- PvmiMIOControl interface allows other PV modules to issue control commands to MIO.
- PvmiCapabilityAndConfig interface facilitates capability and configuration exchange between other PV modules and MIO.
- PvmiMediaTransfer interface defines methods for media data transfer to and from MIO.
- PvmiClockExtensionInterface interface (used only for rendering) allows other PV modules to provide OsclClock object to MIO component, which is necessary for synchronization of media rendering. It is optional for an MIO component to implement this interface. If the MIO component implements this interface, PVPlayer Engine will defer the responsibility of time synchronization of media rendering to the MIO component. Typically, all MIO components that receives compressed media data types and uses decoding functionality provided by hardware components under the MIO should implement this interface.
- If an MIO component implements the optional PvmiClockExtensionInterface (used only for rendering), it should also implement the OsclClockStateObserver interface that allows MIO to receive notification when OsclClock state is changed. It is necessary for the MIO to receive such notifications to handle time synchronization of media rendering.
- The HWObserver interface would define the callbacks that the hardware interface would use to interact with MIO component e.g. MDevSoundObserver for Symbian OS.

The class diagram below illustrates the relationship between MIO component and the various PVMI interfaces that it implements

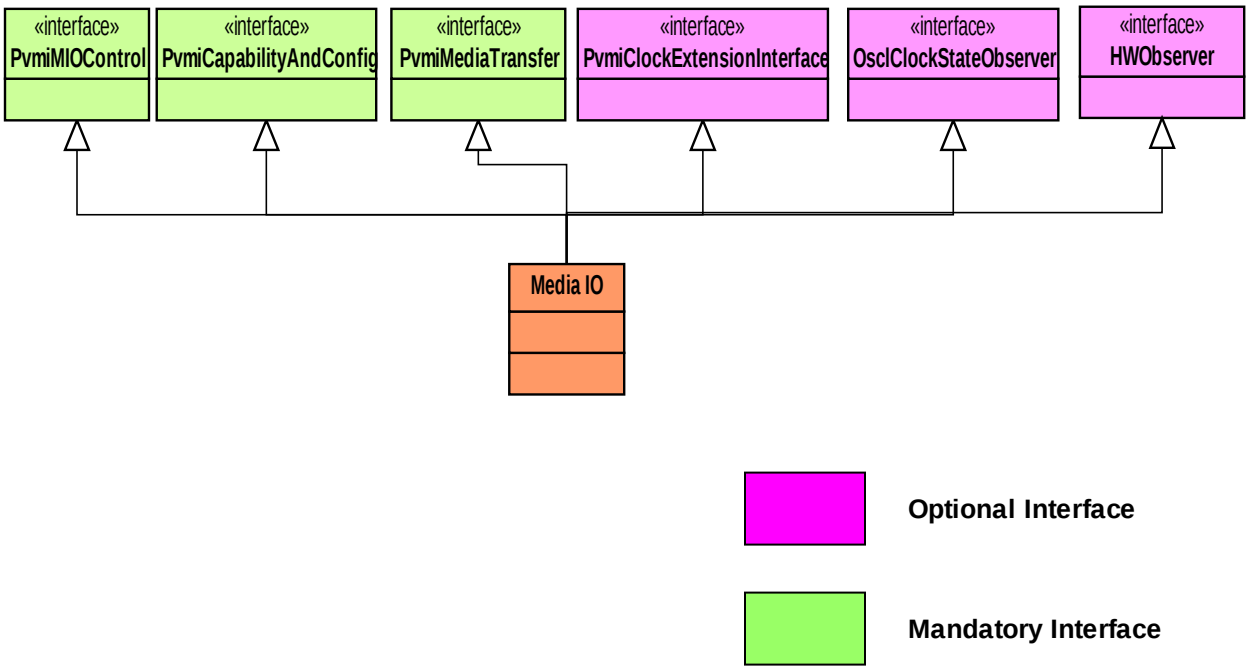

Figure 2: Diagram of interfaces implemented by MIO components.

2.3 Active and Passive MIO Components

If an MIO component implements the optional PvmiClockExtensionInterface, the multimedia framework will defer the responsibility of time synchronization of media rendering to the MIO component, it is called an active MIO component. An active MIO component should also implement another interface OsclClockStateObserver. The multimedia framework will query the MIO component for the PvmiClockExtensionInterface, and if it implements that interface, the MIO will given a reference to the common clock using those APIs. The MIO is considered active at that point and responsible for handling rendering synchronization.

If an MIO component does not implement the optional PvmiClockExtensionInterface, PV SDK will take the responsibility of time synchronization of media rendering to the MIO component, it is called a passive MIO component. A passive MIO component should not implement interface OsclClockStateObserver.

Because of the differences in responsibilities for active versus passive MIOs, the detailed interactions between the MIO component and the multimedia framework can be different in certain use-cases.

2.4 Capability and Configuration Exchange

To allow playback and capturing of a wide range of media types with different properties on various hardware platforms with different capabilities, it is necessary for the MIO to report its capabilities to other PV modules and to receive the necessary properties of the media data. To facilitate this information exchange, MIO components need to implement PvmiCapabilityAndConfig interface, and return a pointer to the implementation when requested in QueryInterface API call. The PVPlayer and PVAuthor specific capability exchange is explained in Section 7.1 and 7.2 respectively.

The PvmiCapabilityAndConfig interface defines a flexible way to configure any general capability and settings between two modules. For the MIO component, based on the current usage of PvmiCapabilityAndConfig interface by other PV modules, it only needs to implement support for some APIs defined in that interface. More specifically, all current usage of PvmiCapabilityAndConfig is restricted to the synchronous calls of the interface, which the MIO component must implement. The asynchronous APIs are not currently used.

The capability and configuration settings are identified by unique key strings and packaged as PVMI key-value pair (PvmiKvp) data structures. The PacketVideo Extended MIME String (PvXms) format is used to specify the key strings. PvXms extends the standard MIME string format by allowing additional levels of subtype strings all separated by the slash character. Typically, the MIO component can use the predefined extended MIME strings in the pvmi_kvp.h header file and use string comparison to identify the capability being set or queried. Optionally for other strings that are not predefined, the query can be identified by parsing according to the PvXms syntax.

2.5 Media Transfer

The primary functionality of an MIO component in PVSDK architecture is to be the data sink or data source for/of media data. The PVMI framework defines the PvmiMediaTransfer interface to facilitate this exchange of media data. The exchange of media data and commands occurs between two peer modules implementing PvmiMediaTransfer interface. On one side is PV SDK datapath, and the other is the MIO component. Besides media data, some “in-data” commands such as End of Data notification and error events are also exchanged through PvmiMediaTransfer interface.

2.6 Event Reporting

To report unsolicited error or information events from MIO component to PV SDK datapath, the MIO component should call the peer’s writeAsync function. To indicate to the peer that the writeAsync contains event information, the MIO component should call writeAsync with the following parameters

Parameter	Value
`format_type`	`PVMI_MEDIAXFER_FMT_TYPE_NOTIFICATION`
`format_index`	`PVMI_MEDIAXFER_FMT_INDEX_ERROR_EVENT` for error events `PVMI_MEDIAXFER_FMT_INDEX_INFO_EVENT` for info events
`data`	Pointer to a `PVMFAsyncEvent` object. If extended errors are defined

	for the MIO component, a `PVMFBasicErrorInfoMessage` object should be created to contain the extended error event. The appropriate constructor of `PVMFAsyncEvent` should be used to pass the `PVMFBasicErrorInfoMessage` object as `aEventExtInterface` parameter.
`data_len`	Size of `PVMFAsyncEvent` object

PV SDK datapath would call the MIO component's writeComplete function after the event is processed. If the event data is allocated dynamically, it should be deallocated upon writeComplete is received.

Not all the events are sent to application. Only the events with following EventType (a member of `PVMFAsyncEvent`) are sent to application:

For PVMI_MEDIAXFER_FMT_INDEX_ERROR_EVENT

PVMFErrCorrupt	PVMFErrOverflow	PVMFErrUnderflow
PVMFErrTimeout	PVMFErrNoResources	PVMFErrResourceConfiguration
PVMFErrResource	PVMFErrNoMemory	PVMFErrProcessing,

For PVMI_MEDIAXFER_FMT_INDEX_INFO_EVENT for info events

PVMFInfoDataDiscarded	PVMFErrNoResources	PVMFSuccess

The sequence diagram in Figure 3 below illustrates how an error from the hardware can propagate up to the application level.

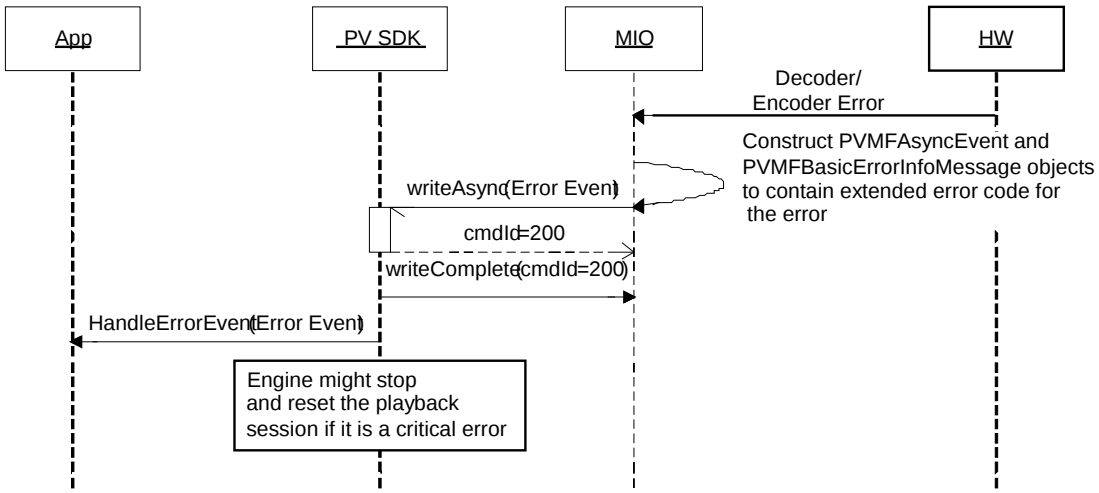

Figure 3: Sequence diagram showing the propagation of error events.

3 Role in the PVMF Architecture

3.1 Overall Architecture of PVPlayer

MIO components are used as data sinks in PVPlayer SDK architecture. The MIO component sits between the media rendering hardware and the rest of PVPlayer SDK modules, and act as a glue layer between the two.

Figure 4 illustrates how the MIO component interacts with the components around it.

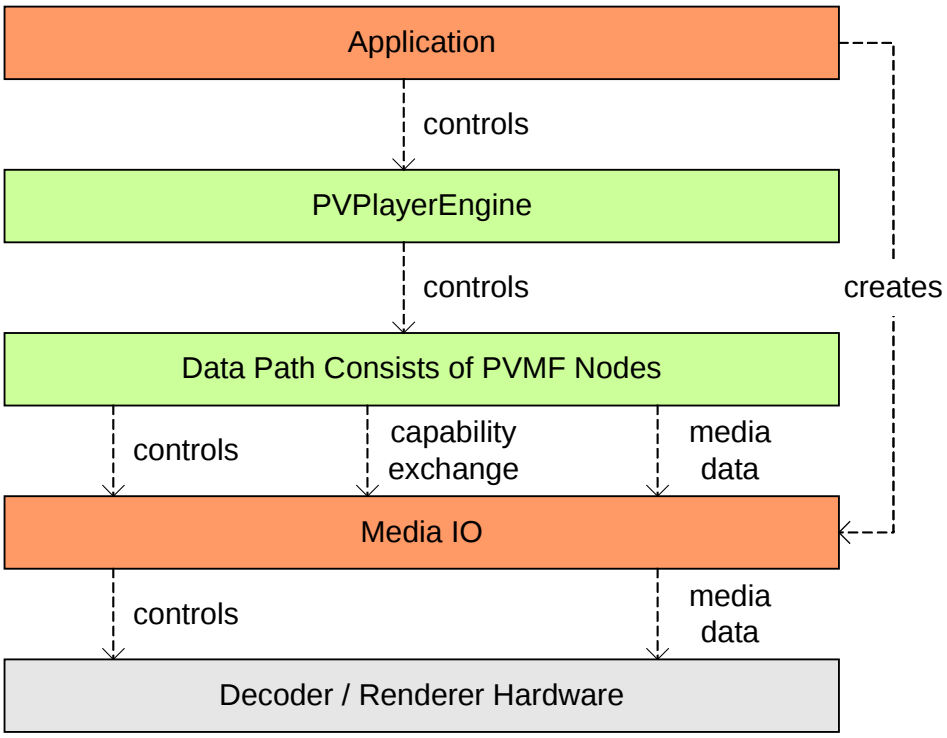

Figure 4: A diagram of the interaction between MIO components and player components.

3.2 Overall Architecture of PVAuthor

MIO components are used as data source in PVAuthor SDK architecture. The MIO component sits between the media capturing devices/hardware and the rest of PVAuthor SDK modules, and act as a glue layer between the two. Figure 5 shows the High level relationship between MIO components and PVAuthor SDK.

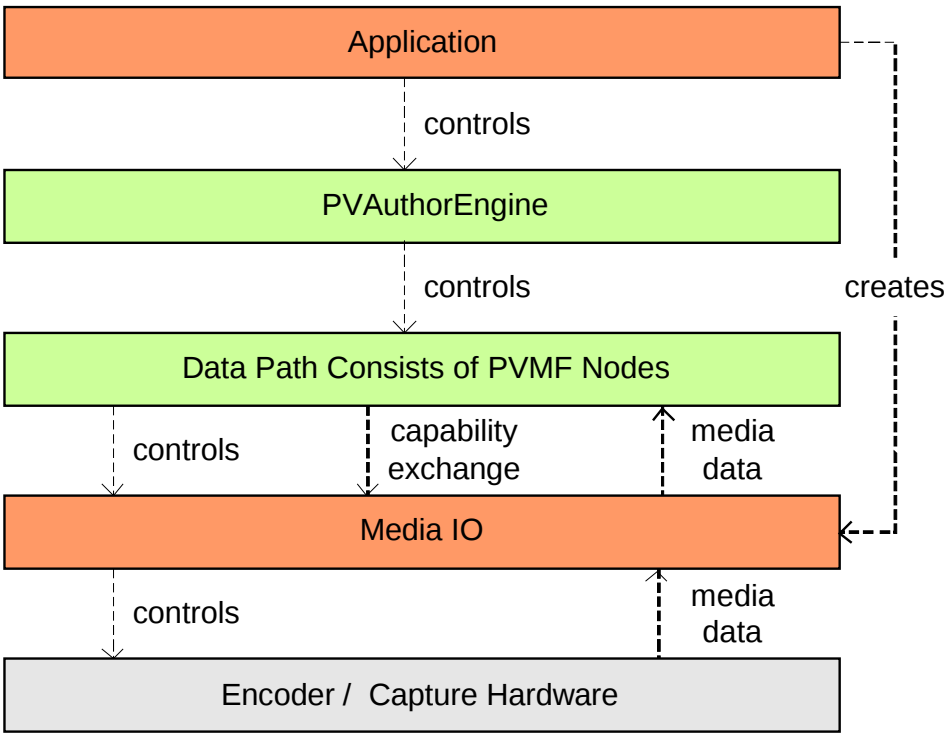

Figure 5: A diagram of the interaction between MIO components and author components.

Figures 6 and 7 illustrate how MIO component interacts with author components in more detailed diagrams showing the structure of the media data-flow graph. The two cases of compressed and uncompressed input sources are shown. In the case of uncompressed input encoders are part of the media data processing in the data-flow graph, while for the compressed input the encoders are not needed. Note that it is not necessary for both inputs to be the same in terms of providing compressed or uncompressed data, so for example, the audio could be compressed while the video is uncompressed. Each MIO component is treated independently in this sense.

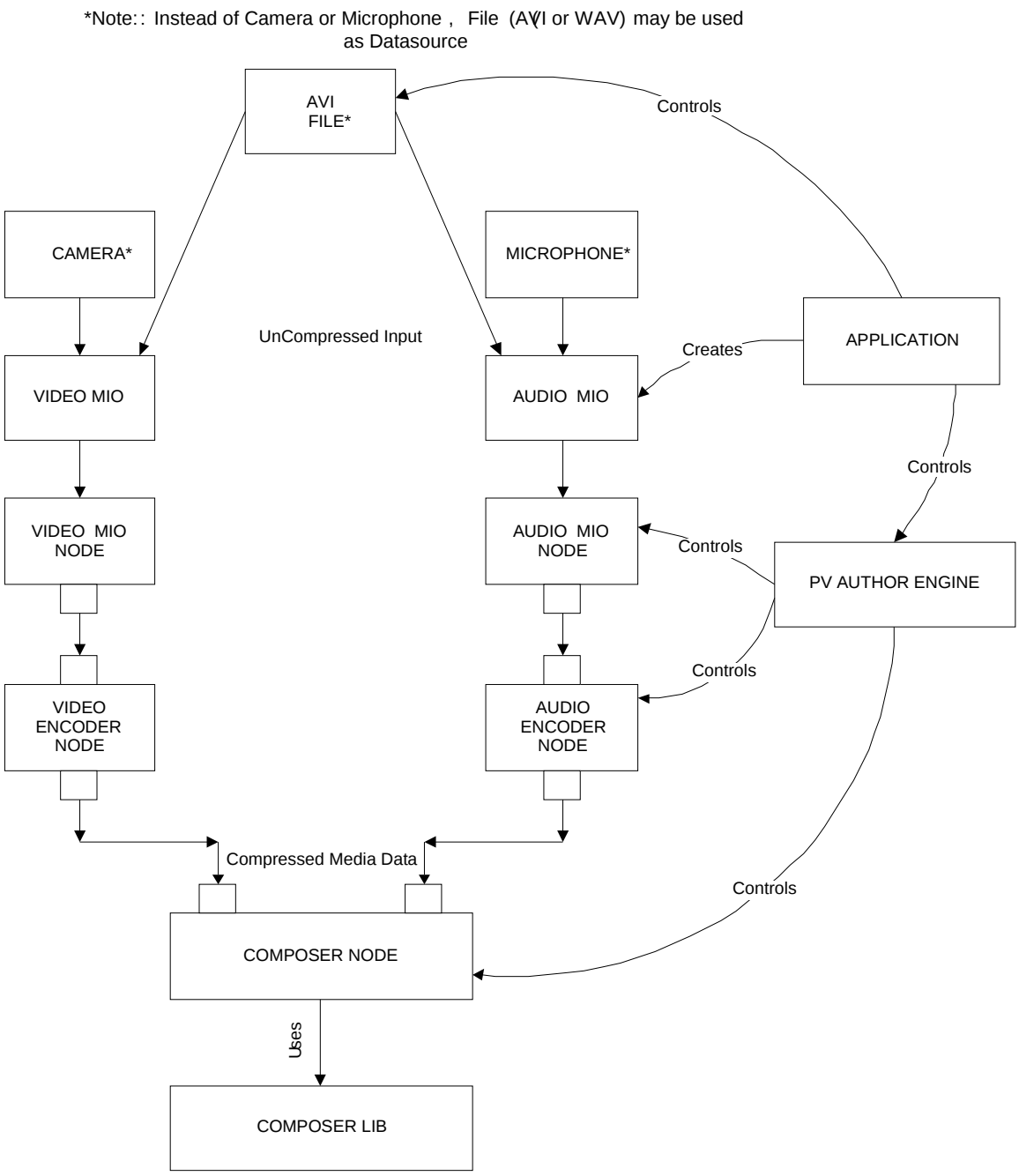

Figure 6: Author flow graph with uncompressed input.

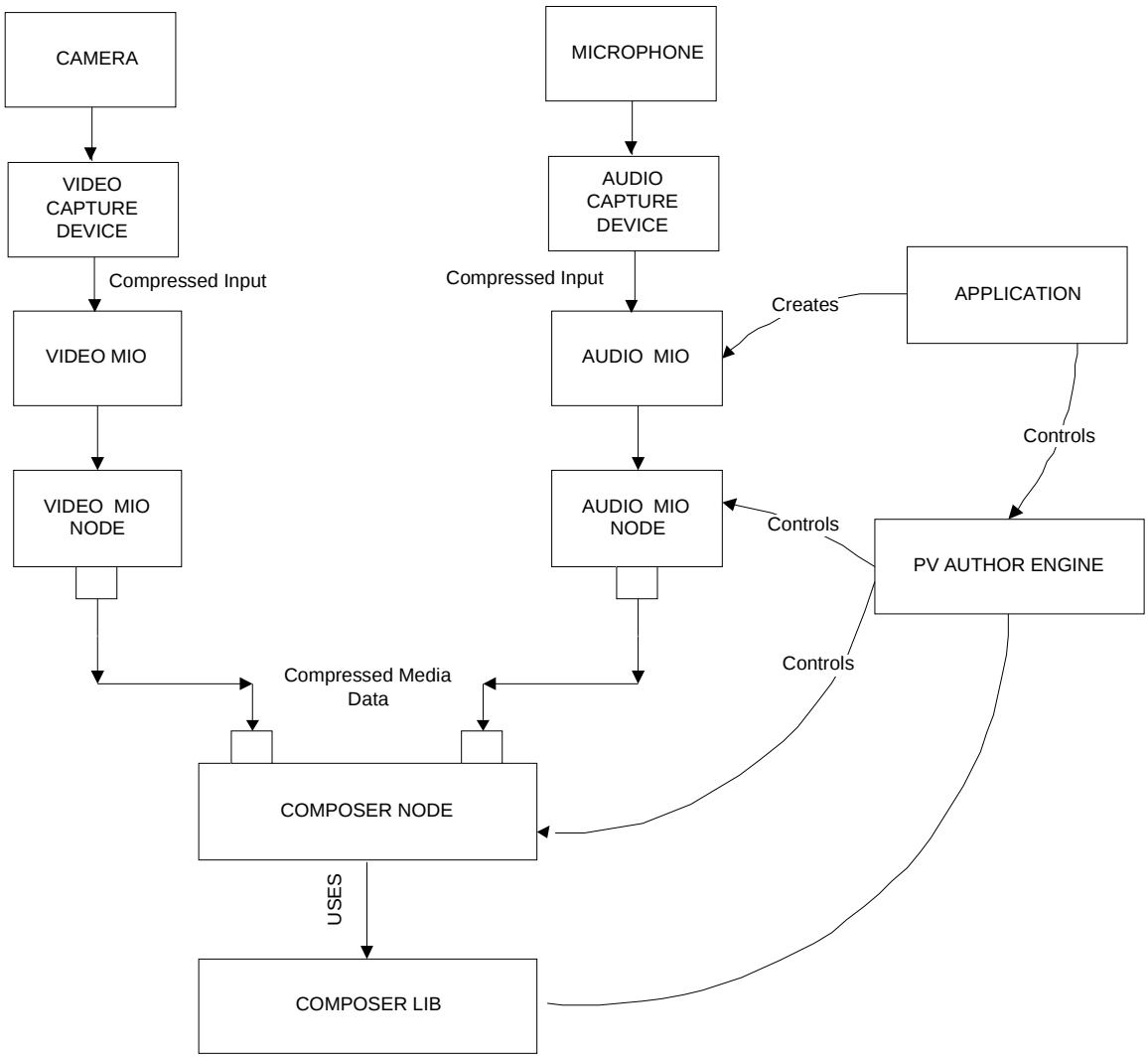

Figure 7: Author flow graph with compressed input.

4 PVPlayer APIs

4.1 PVPlayer AddDataSink

Under PVPlayer SDK architecture, the application is responsible for creating the MIO component, and providing the MIO component as a data sink to PVPlayer using PVPlayerInterface::AddDataSink API. AddDataSink API itself does not perform any additional action on MIO component or the hardware. It simply adds the data sink to the list of available data sinks. Upon the successful completion of AddDataSink command from player engine, the data sink becomes available for other PV modules to connect to for media playback.

4.2 PVPlayer Prepare

Under PVPlayer SDK architecture, the bulk of initialization and configuration of MIO component is done during the processing of PVPlayerInterface::Prepare. The below sequence diagram illustrates the interaction with an MIO component during this prepare phase.

4.2.1 MIO connect

Connect is a synchronous API inherited from PvmiMIOControl interface. PVPlayer datapath calls this function to establish a control session with the MIO component. The caller in this API call provides an observer object, and the MIO component should use it for making callbacks to complete control commands.

4.2.2 MIO QueryInterface

QueryInterface is an asynchronous API inherited from PvmiMIOControl interface. Other PV modules may call this API to retrieve extension interface supported by the MIO. As discussed in section 8, MIO component must implement PvmiCapabilityAndConfig and optionally PvmiClockExtensionInterface extension interface (please refer section 8). When this API is called, MIO component needs to provide a pointer to an instance of the implementation of the requested interface, which other PV modules will use later on. The MIO component should call RequestCompleted callback to complete the call.

4.2.3 MIO Init

Init is an asynchronous API inherited from PvmiMIOControl interface. MIO should implement this function to create an instance to hardware driver, and reserve and initialize the hardware driver as necessary. At the time of Init call, properties and format specific information of the media to be played should have already been provided through the capability and configuration exchange that was done before Init. The MIO component should use this information to initialize and reserve the hardware for playback usage by the MIO component.

Init is an asynchronous API, and the MIO component should call RequestCompleted callback to complete the call after processing is completed.

4.2.4 MIO Start

Start is an asynchronous API inherited from PvmiMIOControl interface. MIO should prepare itself and the hardware to receive media data that will arrive soon after Start is completed. Actions such as memory allocation and requesting hardware driver for shared memory buffers should be done at this stage. Note that this Start call is not meant to signal the immediate start of media rendering because media data is not yet available to the MIO component or hardware at this point.

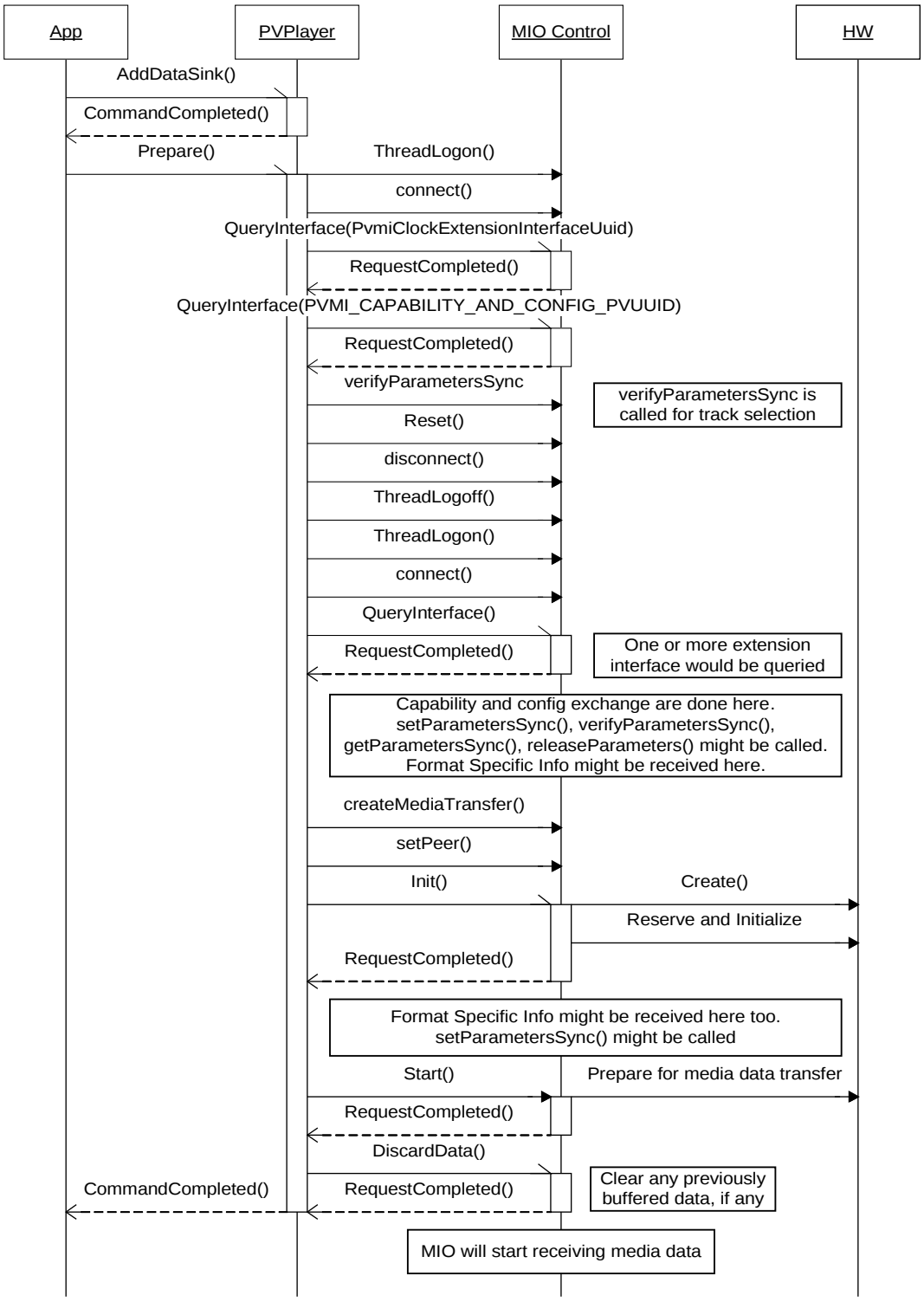

Figure 8: Sequence diagram of the player prepare command.

The MIO component should call RequestCompleted callback to complete the call after processing is completed. The MIO component would need to complete the Start call before incoming media data flow into the MIO component starts. Handling of incoming media data will be discussed in Section 38 of this document.

Please note that the DiscardData() call may come before or after the Start(). As mentioned in the next section, no assumption should be made about the order.

4.2.5 MIO DiscardData

DiscardData is an asynchronous API inherited from PvmiMIOControl interface. This API is called during PVPlayerInterface::Prepare handling to clear any previously buffered data before playback starts, and when repositioning is requested from end user. There are two versions of DiscardData API defined in PvmiMIOControl interface, one without a specific timestamp and one with a timestamp. If a timestamp is provided in DiscardData API call, the MIO component should not render data up to the requested timestamp; otherwise, all currently queued data in MIO component should not be rendered. Please note that each data sample can have a duration associated with it. If timestamp of data plus duration is less than or equal to the requested timestamp, the sample should not be rendered.

Even for the media data should not be rendered, they should also be released by call the peer's writeComplete function. It is necessary to call the peer's writeComplete function to complete the appropriate pending writeAsync calls queued by the MIO component and hardware, and clear the appropriate memory buffers containing copied media data from previous writeAsync calls. The hardware under MIO component is required to support external request to clear its internal buffers to support this functionality. Furthermore, it is highly recommended that hardware should support clearing buffered data based on timestamp to support repositioning use-cases.

The MIO component should call RequestCompleted callback to complete the call after processing is completed. The MIO component would need to complete the DiscardData call before rendering any data or processing incoming data from the newly requested position.

The DiscardData request may happen at any time after init is complete. The MIO component should not the DiscardData request even if there is no data to discard; it should simply return success in that case.

4.3 PVPlayer Start

When application issues Start command to PVPlayer, the datapath should have already been prepared from the Prepare command, and is ready to start rendering media data. The interaction with MIO component upon PVPlayer start is different for active/passive MIO component. About active/passive MIO component, please refer to section 2.3.

If the MIO component is a passive MIO component, incoming media data flow would start after PVPlayer enters started state, and the MIO component and hardware should render the data as soon as possible.

If the MIO component is an active MIO component, a callback call OsclClockStateObserver::ClockStateUpdated() would be called to notify start of the playback clock. This start of playback clock signals the start of media rendering, and the MIO component should request the hardware to start rendering as soon as possible. Additional details about the timing and synchronization of rendering will be discussed in Section 54.
More about the Media Transfer please refer to section 38.

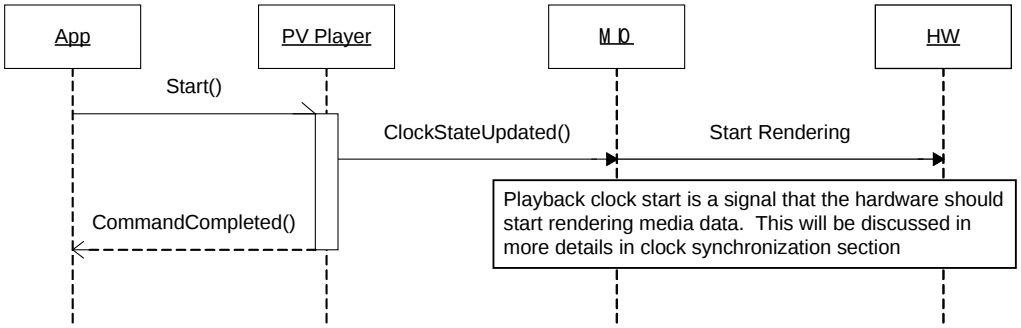

Figure 9: Sequence diagram showing active MIO interaction with clock at start of playback.

4.4 PVPlayer SetPlaybackRange

When application requests repositioning by calling PVPlayerInterface::SetPlaybackRange API, PVPlayer datapath needs to discard all buffered data along the datapath before starting to process data from the newly requested position. Therefore, DiscardData API would be called to MIO, and MIO needs to clear all buffered data in the MIO and in the hardware. Please refer to section 18 for details about MIO DiscardData implementation. After DiscardData, the data source would be repositioned to the newly requested position, and start propagating data from the new position along the datapath. The MIO would need to appropriately prepare the hardware to play the data from new position. The active MIO might also need to interact with the playback clock to synchronize it with the start of rendering of new data (it is not showed in this sequence). Please refer to section 54, for details about clock synchronization for repositioning.

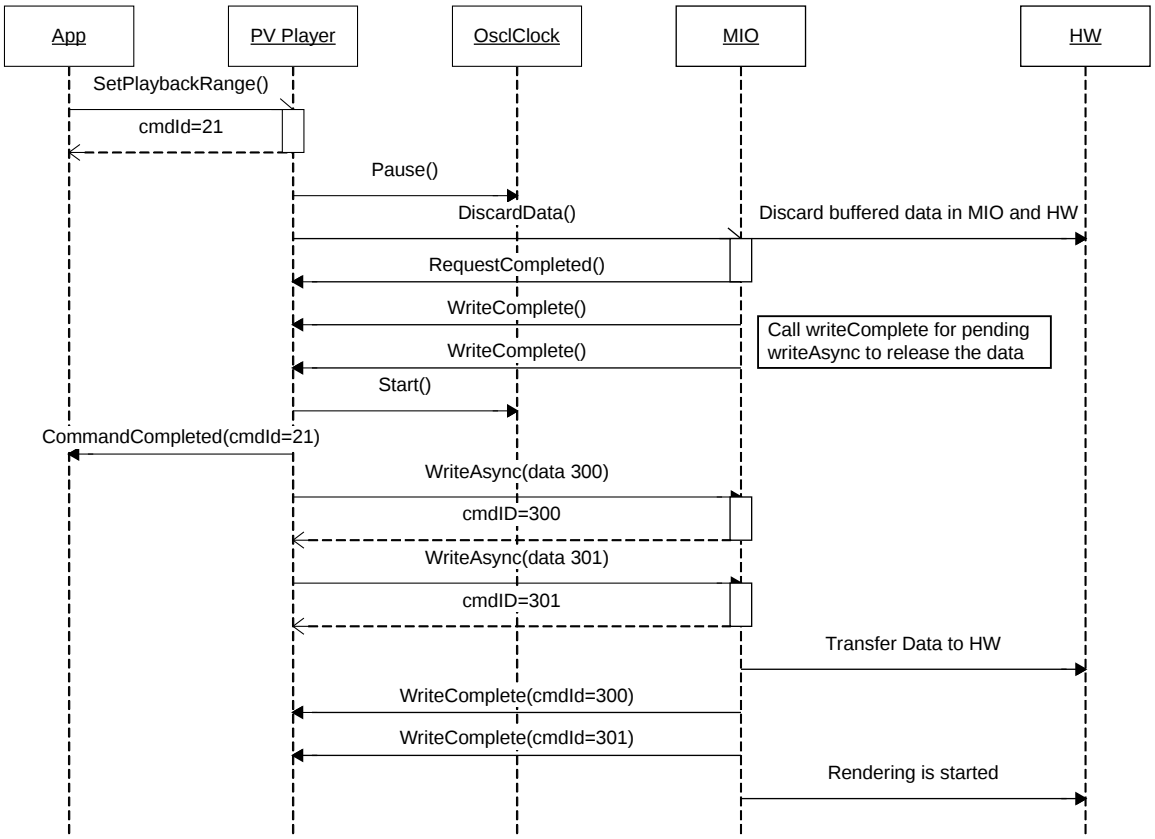

Figure 10: Sequence diagram of repositioning use-case.

4.5 PVPlayer Pause – Resume

When the application issues pause command to PVPlayer, the MIO component Pause API would be called. The MIO component should request the hardware to pause playback.

For a passive MIO component, PVPlayer datapath will not send data to the MIO component at pause state or while the clock is pausing. For an active MIO component, even during the paused state or while the clock is pausing, the PVPlayer datapath will continue sending data to the MIO component, and MIO component should continue to buffer the received data. The MIO component can reject the new data if its internal buffers are full. Any data buffered should be held until playback is resumed, and rendering would resume at that point. The active MIO might also need to interact with the playback clock to do the pause/ resume exactly when clock pause/ resume (it is not shown in this sequence). For more details about active versus passive MIO components, refer to Section 2.3.

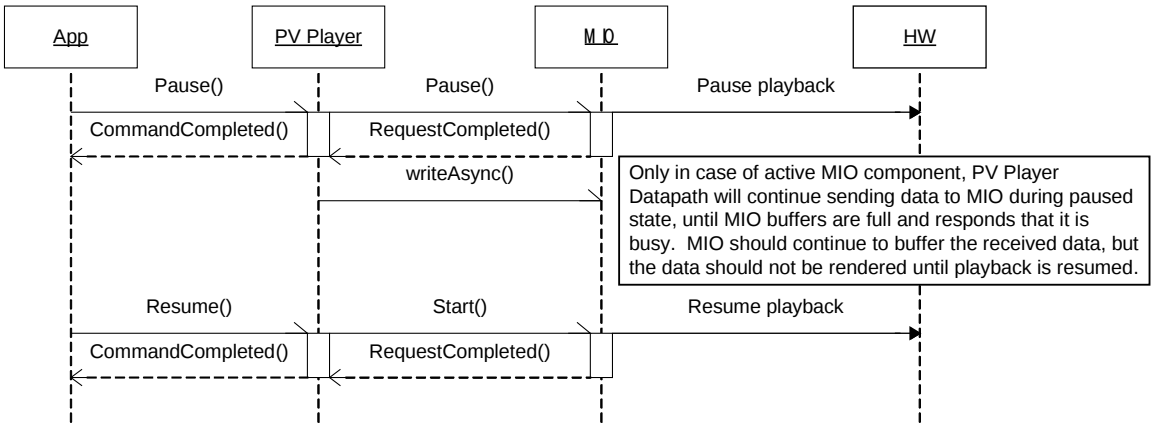

Figure 11: Sequence diagram of the pause/resume use-case.

4.6 PVPlayer Stop and Reset

When application issues Stop command to PVPlayer, playback is stopped and the datapath is torn down. The MIO should request hardware to stop playback, complete all pending media data messages in buffer, and release hardware resources at this time. After Stop command is completed, Application would call Reset to complete the closure of playback session.

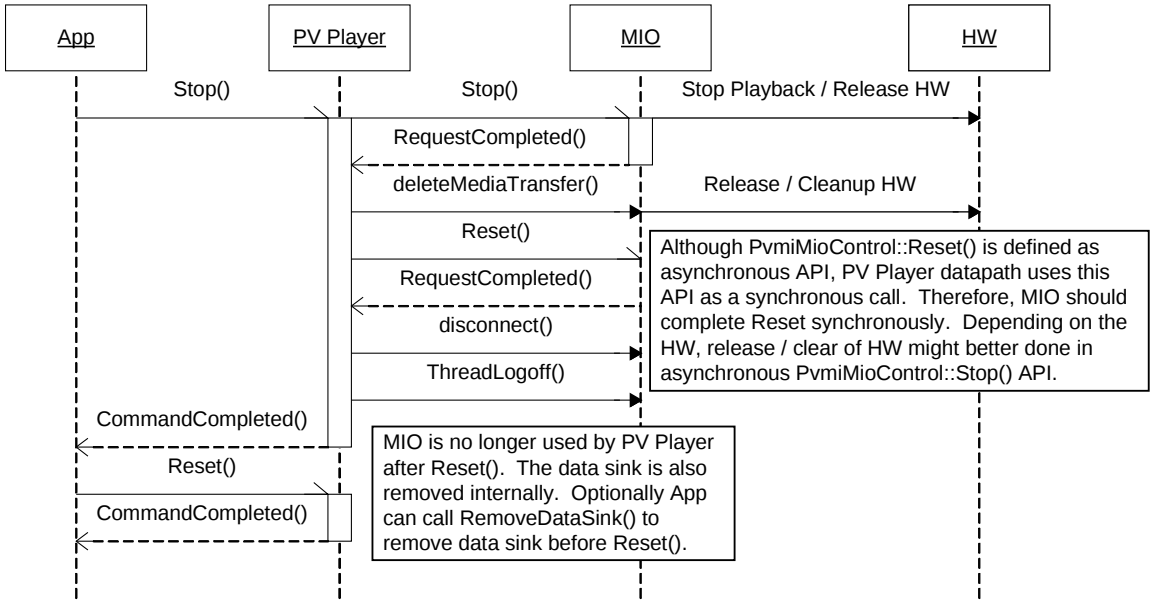

Figure 12: Sequence diagram of stop and reset.

4.6.1 MIO Stop

Stop is an asynchronous API inherited from PvmiMIOControl interface. Upon receiving this request, the MIO should request the hardware to stop playback and release any buffers. Furthermore, the MIO component should release any buffered media data back to the PVPlayer datapath by calling the writeComplete callback (this will be explained in further details in Section 38). The RequestCompleted callback should be called after all actions described above are completed.

4.6.2 MIO Reset

Reset is an asynchronous API inherited from PvmiMIOControl interface. The Reset method may be called in any state, so the component **must never return InvalidState**. The MIO component should stop any playback, release any buffers, and clean up all resources and state. If the MIO has not been initialized then there may not be any cleanup work, so it can trivially return success in this case. In general, it is expected that the Reset method should not fail. It will certainly be considered a fatal error, and it would be best to modify the MIO to return success if possible.

4.7 PVPlayer Cancel Commands

The application can cancel all or a particular command through PVPlayerInterface. If the command to be cancelled requires cancelling of a queued command of MIO component, the MIO `CancelCommand` or `CancelAllCommands` API would be called. Since cancel command has a higher priority comparing with other control commands, it should be added at the front of the command queue. If the MIO component is currently processing a command and pending its completion, the MIO component should complete the command in process first, and then process the cancel command next.

4.7.1 MIO CancelCommand

CancelCommand is an asynchronous API inherited from PvmiMIOControl interface. The MIO component should search through its command queue to locate the command requested to be cancelled, and remove it from the command queue.

4.7.2 MIO CancelAllCommands

CancelAllCommands is an asynchronous API inherited from PvmiMIOControl interface. The MIO component should remove all pending commands from the command queue asynchronously. The sequence diagram below illustrates how MIO component should handle this command

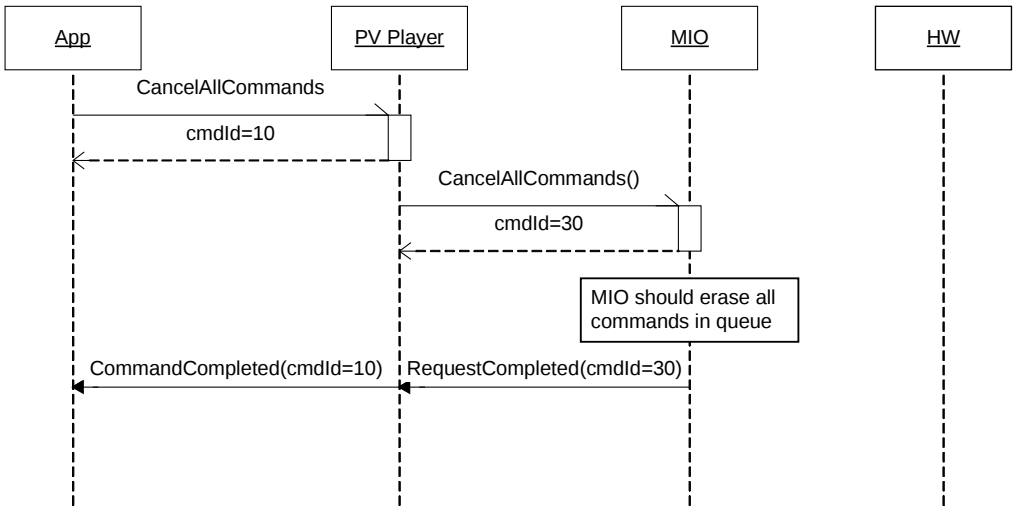

Figure 13: Sequence diagram of CancelAllCommands.

5 PVAuthor APIs

The APIs invoked from the application and the corresponding activities that are performed on the MIO end are depicted below via the sequence diagram that follows. Explanation of the same follows in section 23 to 25 below:

5.1 PVAuthorEngine AddDataSource

Under the PVAuthor architecture, the application creates media data source objects and provides them to PVAuthor Engine through the PVAuthorEngine::AddDataSource method. The data source is integrated to the PVAuthor engine by using media I/O interface and the media input node. The AddDataSource API itself does not perform any additional action on the MIO component or the hardware. It simply adds the data source to the list of available data sources. Upon the successful completion of AddDataSource command from author engine, the data source becomes available for PV modules to connect for media capturing.

5.1.1 MIO connect

Connect is a synchronous API inherited from PvmiMIOControl interface. PVAuthor datapath calls this function to establish a control session with the MIO component. The caller in this API call provides an observer object, and the MIO component should use it for making callbacks to complete control commands.

5.1.2 MIO QueryInterface

QueryInterface is an asynchronous API inherited from PvmiMIOControl interface. Other PV modules may call this API to retrieve extension interface supported by the MIO. As discussed in section 8, MIO component must implement PvmiCapabilityAndConfig. When this API is called, MIO component needs to provide a pointer to an instance of the implementation of the requested

interface, which other PV modules will use later on. The MIO component should call RequestCompleted callback to complete the call.

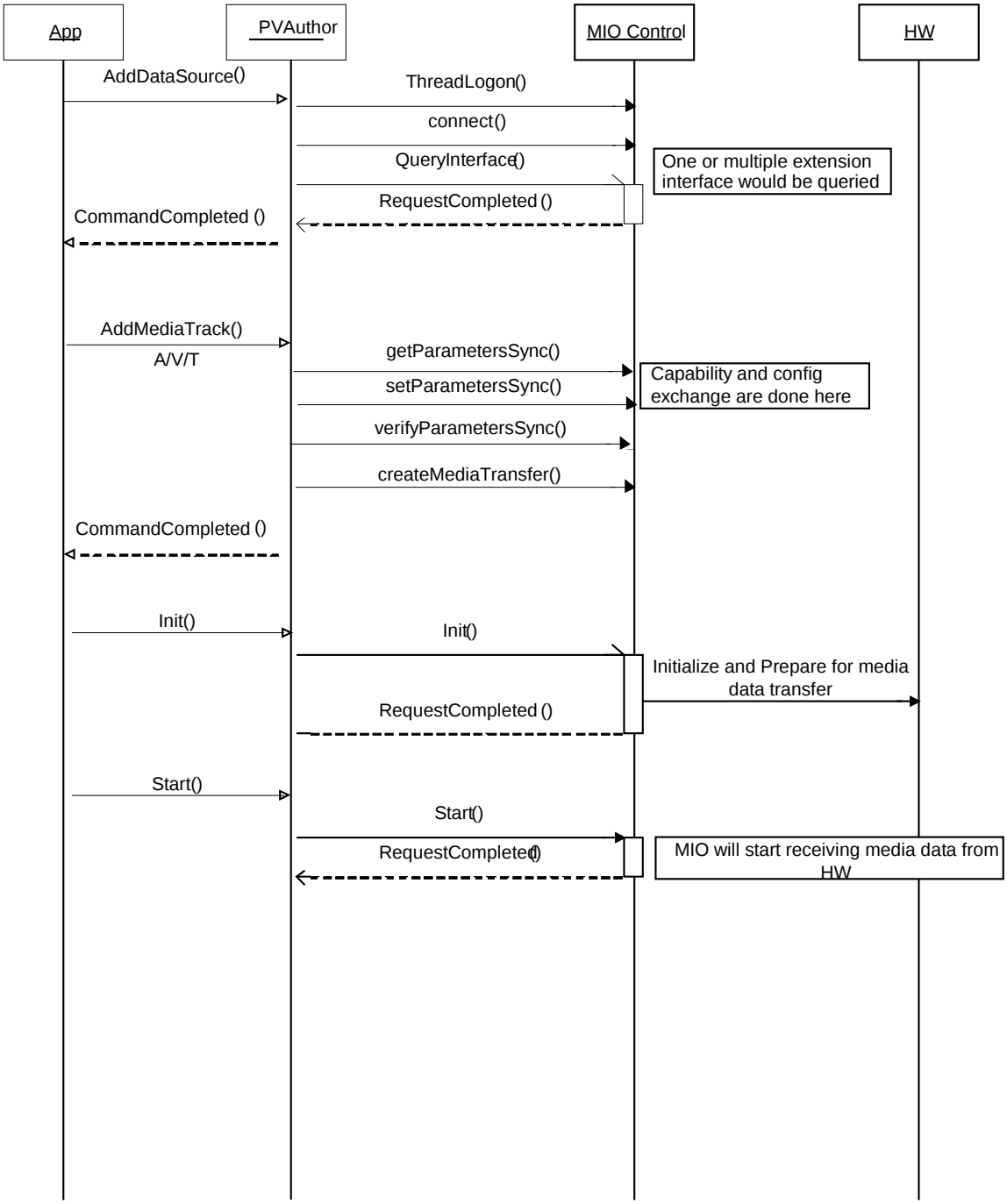

Figure 14: Sequence diagram of PVAuthor and MIO component initialization.

5.2 PVAuthorEngine AddMediaTrack

To create a file with multiple media tracks, the client will need to call AddMediaTrack for each of the track, for example AMR audio with H263 video and text tracks. Adding of media track is done through the AddMediaTrack method. The client will need to specify the input PVMF node that provides the source data for this media track, the Mime type of the encoder to be used to encode the source data, and the file format composer in which a media track is added. At the MIO level, Capability and config exchange is done during this time and it is decided whether the encoder would be required or not, depending upon whether the data is compressed or uncompressed. If data is uncompressed the encoder would be required otherwise encoder can be eliminated from datapath.

5.3 PVAuthorEngine Init

5.3.1 MIO Init

Init is an asynchronous API inherited from PvmiMIOControl interface. MIO should implement this function to create an instance to hardware driver, and reserve and initialize the hardware driver as necessary. At the time of Init call, properties and format specific information of the media to be captured should have already been provided through the capability and configuration exchange that was done before Init. The MIO component should use this information to initialize and reserve the hardware for capturing usage by the MIO component. For compressed input, the init phase is used to get format-specific information such as decoder configuration information. The MIO component should call RequestCompleted to signal that processing has been completed.

5.4 PVAuthorEngine Start

5.4.1 MIO Start

Start is an asynchronous API inherited from PvmiMIOControl interface. After its completion, this API triggers the media data transfer from the hardware to the MIO component. The MIO component then sends the data to the datapath by calling writeAsync of the media input node. The detailed description of the process of receiving the media data from hardware and passing it on to the PVAuthor datapath is given in section 8.2. If the data from the MIO component is compressed, the encoder is not needed so the media input node is connected directly to the composer node within the datapth in that case. For uncompressed input, the media input node provides data to the encoder node, which is connected composer node within the datapath.

In the start phase, the composer node may use the API ‘GetInputParametersFromPeer’ to query information such as the sampling rate, bitrate etc. In this case, getParametersSync is called on the MIO component before the data transfer to the composer node begins.

The MIO component should call RequestCompleted after processing the Start request to signal it finished the request. Once RequestCompleted is called, the MIO component starts sending data to the datapath.

5.5 PVAuthor Pause – Resume

When the application issues pause command to PVAuthor, the MIO component Pause API would be called. The MIO component should request the hardware to pause capture. During the paused state, the hardware would not send any data to the MIO, hence there would be no exchange of media data between MIO component and PVAuthor datapath. When the application issues resume command to PVAuthor, the MIO component start API would be called. In the start API, to resume recording, the MIO should ask the hardware interface to capture more data by calling the appropriate API (e.g., RecordData in case of Symbian OS). In order to resume the capture of data, there is no need to initialize the hardware again; calling the API to resume the capture would suffice.

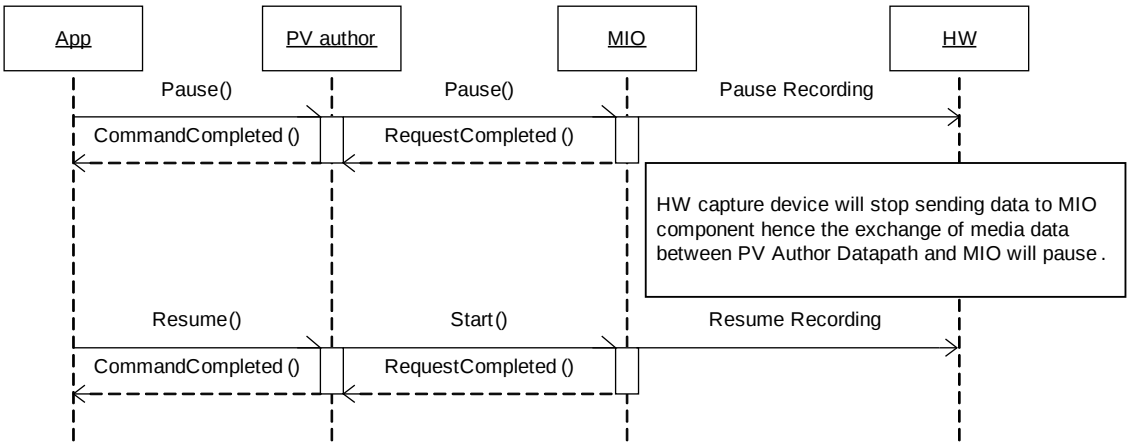

Figure 15: PVAuthor and MIO interaction for pause and resume.

5.6 PVAuthor Stop

When application issues Stop command to PVAuthor, capturing is stopped and the datapath is torn down. The MIO should request hardware to stop capturing, complete all pending media data messages in buffer, and release the hardware resources at this time.

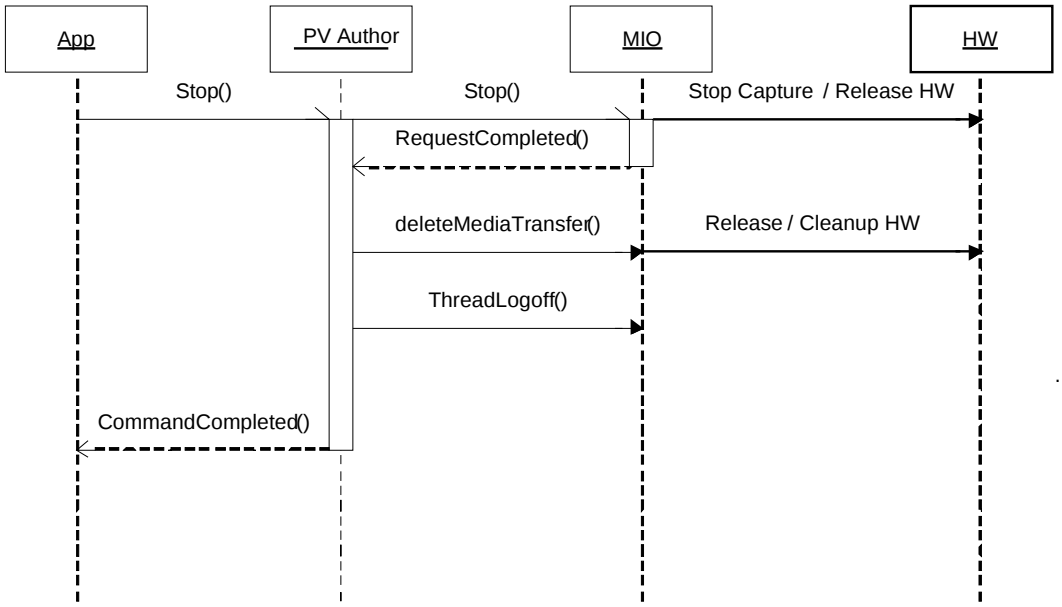

Figure 16: PVAuthor and MIO interaction for stop.

5.6.1 MIO Stop

Stop is an asynchronous API inherited from PvmiMIOControl interface. Upon receiving this request, the MIO component should request the hardware to stop capturing and release resources. Furthermore, the media input node should release any buffered media data by calling writeComplete callback on MIO component (this will be explained in further details in Section 38). RequestCompleted callback should be called after all actions described above are completed.

5.7 PVAuthor Cancel Commands

The application can cancel all or a particular command through PVAuthorEngine. If the command to be cancelled requires cancelling of a queued command of MIO component, the MIO `CancelCommand` or `CancelAllCommands` API would be called. Since cancel command has a higher priority comparing with other control commands, it should be added at the front of the command queue. If the MIO component is currently processing a command and pending its completion, the MIO component should complete the command in process first, and then process the cancel command next.

5.7.1 MIO CancelCommand

CancelCommand is an asynchronous API inherited from PvmiMIOControl interface. The MIO component should search through its command queue to locate the command requested to be cancelled, and remove it from the command queue.

5.7.2 MIO CancelAllCommands

CancelAllCommands is an asynchronous API inherited from PvmiMIOControl interface. The MIO component should remove all pending commands from the command queue asynchronously. The sequence diagram below illustrates how MIO component should handle this command.

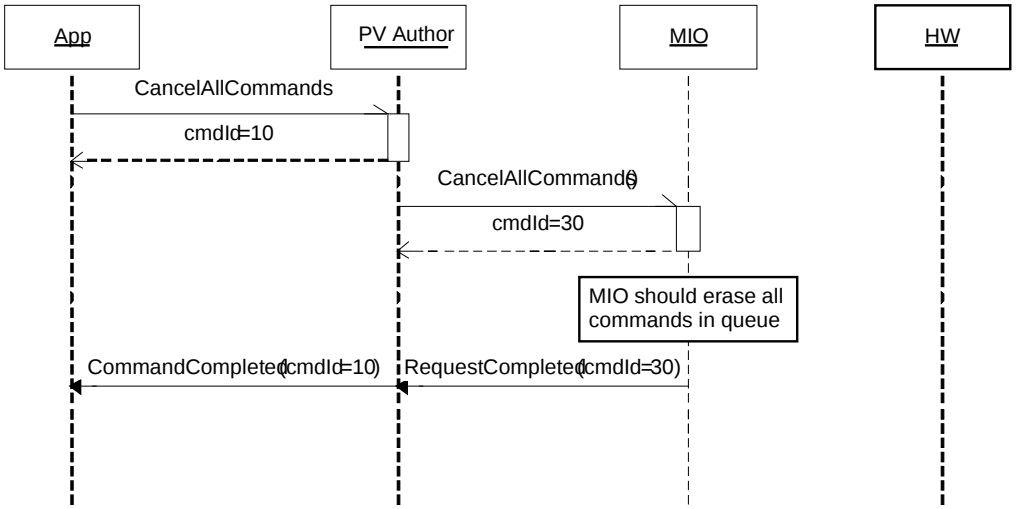

Figure 17: PVAuthor and MIO interaction for CancelAllCommands.

6 Common Interface API's.

The PVMF node design should allow for multi-thread usage scenarios where the node may be created and destroyed in one thread, but used in another thread. The ThreadLogon and ThreadLogoff APIs are the entry and exit points of the **usage thread**. The node can assume that all API calls that occur between ThreadLogon and ThreadLogoff come from the same thread—the primary usage thread.
The thread logon/logoff mechanism allows for separation of the thread-specific initialization and cleanup operations from the construction and destruction operations. Thread-specific operations are like adding active objects to the scheduler, doing memory profiling etc. The state machine reflects the difference between a "Created state" and one that is "logged on".

6.1.1 ThreadLogon()

ThreadLogon is a synchronous command. It is the entry point to the mio comp usage thread. The ThreadLogon call must do all thread-specific initialization for the mio such as adding active objects to the scheduler. ThreadLogon must be allowed in the Created state. After successful thread logon, the mio comp should change to some other state to distinguishable from “Created state”

6.1.2 ThreadLogoff()

ThreadLogoff is a synchronous command. It is the exit point for the usage thread. The ThreadLogoff call must do all thread-specific cleanup such as removing active objects from the scheduler. In other words, threadLogoff should un-do any operations done by ThreadLogon. After successful thread logoff, the state should be “Created”. If thread logoff is unsuccessful, the node may transition to Error state, or may remain in the current state.

6.1.3 connect()

The connect() is also a synchronous command. The observer for MIOControl is set in connect() API.

6.1.4 disconnect()

The disconnect() is also synchronous command. The disconnect() implies that no more callback to observer can be made now.

6.1.5 QueryUUID()

The QueryUUID method is an asynchronous call. This API retrieves the UUID or unique identifier of interfaces supported by mio comp. The call has two modes—normal mode and “exact” mode. Normal mode interprets the input MIME string as a prefix and returns a list of UUIDs of all interfaces supported by the node whose MIME string **contains the prefix**. Exact mode returns the UUIDs of interfaces whose MIME strings match the input MIME string exactly. Typically, **MIO comp should implement “exact” mode**.

6.1.6 QueryInterface()

The QueryInterface() method is an asynchronous call. This API retrieves a pointer to the implementation of a specific interface, given its UUID. QueryInterface should not cause any node state change.

6.1.7 createMediaTransfer()

The createMediaTransfer() method is a synchronous call. It creates and initializes mediatransfer instance and returns its pointer.

7 Capability and Configuration Exchange

7.1 Capability and Configuration Exchange for PVPlayer

7.1.1 getParametersSync

The getParametersSync API allows other PV modules to query for the capability and configurations supported by the MIO. In current implementations, all MIOs are required to support INPUT_FORMATS_CAP_QUERY (".../input_formats;attr=cap") capability, and respond with a list of media formats that can be handled by MIO. PVPlayer Engine uses this information to construct the datapath. The function plays an important role in track selection.

7.1.2 releaseParameters

The releaseParameters API goes in pair with getParametersSync function to release the data that is returned by MIO in the previous getParametersSync call. This API is called when the PV module that requested the information, provided in the previous getParametersSync call, has finished using the data. If memory is allocated in the corresponding getParametersSync call, it can be deallocated or reused.

7.1.3 setParametersSync

The setParametersSync method is the key API of PvmiCapabilityAndConfig interface used to configure the MIO component. Other PV modules call this API to set various settings that are required typically to initialize the MIO itself as well as the decoder or renderer hardware. Once the MIO component gets all the parameters it needs for initialization, it should call ReportInfoEvent() with PVMFMIOConfigurationComplete as the event-type. Typically, this message is sent after MIO component receives all required parameters in setParametersSync. But, the MIO component may choose to send this message at a later stage after initialization is complete. The MIO node will not send any media data to MIO component until it receives PVMFMIOConfigurationComplete.

7.1.3.1 Audio-related Capabilities

For an MIO component that supports audio media data types, it is required to support setting of capabilities identified by the key strings described below.

Key String	Purpose

MOUT_AUDIO_FORMAT_KEY ("x-pvmf/audio/render/media_format;valtype=char*")	PVPlayer datapath uses this setting to inform MIO of the audio media data format that MIO is expected to receive after MIO Start is completed. The value provided by the caller is a MIME string of text media data format type defined in pvmf_format_types.h header file.
MOUT_AUDIO_SAMPLING_RATE_KEY ("x-pvmf/audio/render/sampling_rate;valtype=uint32")	PVPlayer datapath uses this setting to inform MIO of the sampling rate of audio data that MIO is expected to receive. The value provided by the caller is a 32 bit integer in unit of Hz.
MOUT_AUDIO_NUM_CHANNELS_KEY ("x-pvmf/audio/render/channels;valtype=uint32")	PVPlayer datapath uses this setting to inform MIO of the number of audio channels in the data that MIO is expected to receive. The value provided by the caller is either 1 for mono, or 2 for stereo.
PVMF_FORMAT_SPECIFIC_INFO_KEY ("x-pvmf/media/format_specific_info;valtype=key_specific_value")	PVPlayer datapath uses this setting to provide media format specific information, such as codec header, that is necessary for initialization of decoder or renderer hardware. MIO is responsible for parsing the format specific information to retrieve all necessary data from it, and configure the hardware accordingly. The parsing is specific for each media format supported by the MIO, and developers should refer to the individual file format's specification for further information on how to perform such parsing.
PVMF_FORMAT_SPECIFIC_INFO_KEY_PCM ("x-pvmf/media/format_specific_info_pcm;valtype=key_specific_value")	PVPlayer datapath uses this setting to provide a structure, the channelSampleInfo structure, containing information on the number of channels, sample rate, bits per sample, buffer size, and number of buffers. The buffer size and number of buffers contain the requested size and number for the buffers needed by the upstream decoder. These are provided to support the case where the MIO will handle buffer allocation for the decoder. Details of this alternative buffer allocation method is described in . The individual keys for the values covered by this structure such as sample rate, channels, etc may be obsoleted in the future after a transition period, but the MIO component should be prepared to handle those individual keys as well as this combined key currently.

For an audio MIO component that supports AAC format, it is also required to support the following capability.

Key String	Purpose

PVMF_FORMAT_SPECIFIC_INFO_PLUS_FIRST_SAMPLE_KEY ("x-pvmf/media/format_specific_info_plus_first_sample;valtype=uint8*")	PVPlayer datapath uses this setting to provide the codec header plus the first media sample to MIO. The first media sample is required to determine the properties of the AAC bitstream, such as sample rate, number of channels, and AAC stream type.

7.1.3.2 Video-related Capabilities

For an MIO component that supports video media data types, it is required to support setting of capabilities identified by the key strings described below.

Key String	Purpose
MOUT_VIDEO_FORMAT_KEY ("x-pvmf/video/render/media_format;valtype=char*")	PVPlayer datapath uses this setting to inform MIO of the video media data format that MIO is expected to receive after MIO Start is completed. The value provided by the caller is a MIME string of text media data format type defined in pvmf_format_types.h header file.
MOUT_VIDEO_WIDTH_KEY ("x-pvmf/video/render/width;valtype=uint32")	PVPlayer datapath uses this setting to inform MIO of the pixel width of video data that MIO is expected to receive. The value provided by the caller is a 32 bit integer in unit of pixels.
MOUT_VIDEO_HEIGHT_KEY ("x-pvmf/video/render/height;valtype=uint32")	PVPlayer datapath uses this setting to inform MIO of the pixel height of video data that MIO is expected to receive. The value provided by the caller is a 32 bit integer in unit of pixels.
MOUT_VIDEO_DISPLAY_WIDTH_KEY ("x-pvmf/video/render/display_width;valtype=uint32")	PVPlayer datapath uses this setting to inform MIO of the pixel width of the display area that MIO is expected to render to. The value provided by the caller is a 32 bit integer in unit of pixels. It is possible that this is different from the value of pixel width of the video data itself. If the MIO or hardware supports scaling of video data, this information should be used to scale the data before displaying accordingly.

MOUT_VIDEO_DISPLAY_HEIGHT_KEY ("x-pvmf/video/render/display_height;valtype=uint32")	PVPlayer datapath uses this setting to inform MIO of the pixel height of the display area that MIO is expected to render to. The value provided by the caller is a 32-bit integer in unit of pixels. It is possible that this is different from the value of pixel height of the video data itself. If the MIO or hardware supports scaling of video data, this information should be used to scale the data before displaying accordingly.
PVMF_FORMAT_SPECIFIC_INFO_KEY ("x-pvmf/media/format_specific_info;valtype=key_specific_value")	PVPlayer datapath uses this setting to provide media format specific information, such as codec header, that is necessary for initialization of decoder or renderer hardware. MIO is responsible for parsing the format specific information to retrieve all necessary data from it, and configure the hardware accordingly. The parsing is specific for each media format supported by the MIO, and developers should refer to codec specification for the media format for further information on how to perform such parsing.
PVMF_FORMAT_SPECIFIC_INFO_KEY_YUV ("x-pvmf/media/format_specific_info_yuv;valtype=key_specific_value")	PVPlayer datapath uses this setting to provide a set of information in the PVMFYuvFormatSpecificInfo0 class detailing the dimensions, specific YUV layout, and buffer size and number of buffers. The buffer size and number of buffers contain the requested size and number for the buffers needed by the upstream decoder. These are provided to support the case where the MIO will handle buffer allocation for the decoder. Details of this alternative buffer allocation method is described in [6] . The individual keys for the values covered by this structure such as sample rate, channels, etc may be obsoleted in the future after a transition period, but the MIO component should be prepared to handle those individual keys as well as this combined key currently.

7.1.3.3 Text-related Capabilities

For an MIO component that supports text media data types, it is required to support setting of capabilities identified by the key strings described below.

Key String	Purpose
MOUT_TEXT_FORMAT_KEY ("x-pvmf/text/render/media_format;valtype=char*")	PVPlayer datapath uses this setting to inform MIO of the text media data format that MIO is expected to receive after MIO Start is completed. The value provided by the caller is a MIME string of text media data format type defined in pvmf_format_types.h header file.

7.1.3.4 Other Capabilities

For an MIO component that supports control the output playback rate, it is required to support setting of capabilities identified by the key strings described below.

Key String	Purpose
MOUT_MEDIAXFER_OUTPUT_RATE "x-pvmf/mediaxfer/output/rate;type=rel;valtype=int32"	PVPlayer datapath uses this setting to inform MIO of the output playback rate that MIO is expected to received after MIO Start is completed. The value provided by the caller is a 32 bit integer that “10000” means normal speed and “30000” means 3 x speeds.

For an MIO that supports allocating buffers for the upstream decoder, the following setting should be supported for the data path to obtain the buffer allocator.

Key String	Purpose
PVMF_BUFFER_ALLOCATOR_KEY "x-pvmf/media/buffer_allocator;valtype=key_specific_value"	PVPlayer datapath uses this in a getParameterSync to query the MIO component for a buffer allocator object. The MIO is not required to supply an allocator but this can be used to allow the renderer to supply buffers to the decoder. See the following document[6] for information.

7.1.4 verifyParametersSync

This API is called by PVPlayer datapath to verify whether the capability specified is supported by the MIO component and the hardware. The key strings listed in section 30 are used to identify the configuration to be verified. This API should return PVMFSuccess if the capability and value specified are supported or PVMFErrNotSupported if they’re not supported.

7.2 Capability and Configuration Exchange for PVAuthor

7.2.1 getParametersSync

The getParametersSync API allows other PV modules to query for the capability and configurations supported by the MIO. .

7.2.1.1 File Format-related Capabilities

An MIO component can be queried for the following capabilities:

Query	Purpose
OUTPUT_FORMATS_CAP_QUERY ("…/output_formats;attr=cap") or OUTPUT_FORMATS_CUR_QUERY ("…/output_formats;attr=cur")	PVAuthor datapath uses this query to ask MIO of the list of media formats that can be provided from the capture component PVAuthor Engine uses this information to construct the datapath. The key returned from MIO would be: OUTPUT_FORMATS_VALTYPE ("…/output_formats;valtype=uint32")
OUTPUT_TIMESCALE_CUR_QUERY ("…/output/timescale; attr=cur")	PVAuthor datapath uses this query to ask MIO about the timescale of the concerned stream. The key returned from MIO would be: OUTPUT_TIMESCALE_CUR_VALUE ("…/output/timescale; valtype=uint32")

7.2.1.2 Video-related Capabilities

For an MIO component that supports video media data types, it is required to support querying of following capabilities.

Query	Purpose
VIDEO_OUTPUT_WIDTH_CUR_QUERY ("…/output/width; attr=cur")	PVAuthor datapath uses this query to ask MIO of the pixel width of video data that MIO will send. The value is a 32 bit integer in unit of pixels. The key returned from MIO would be: VIDEO_OUTPUT_WIDTH_CUR_VALUE ("…/output/width;valtype=uint32")
VIDEO_FRAME_ORIENTATION_CUR_QUERY ("…/output/frame_orientation; attr=cur")	PVAuthor datapath uses this query to ask MIO of the frame orientation of video data that MIO will send. The value provided is an 8 bit integer. The key returned from MIO would be: VIDEO_FRAME_ORIENTATION_CUR_VALUE ("…/output/frame_orientation;valtype=uint8")
VIDEO_OUTPUT_HEIGHT_CUR_QUERY ("…/output/height; attr=cur")	PVAuthor datapath uses this query to ask MIO of the pixel height of video data that MIO will send. The value provided is a 32 bit integer in unit of pixels. The key returned from MIO would be: VIDEO_OUTPUT_HEIGHT_CUR_VALUE ("…/output/height;valtype=uint32")
VIDEO_OUTPUT_FRAME_RATE_CUR_QUERY ("…/output/frame_rate; attr=cur")	PVAuthor datapath uses this query to ask MIO of the frame rate of video data that MIO will send. The value provided is a 32 bit integer The key returned from MIO would be: VIDEO_OUTPUT_FRAME_RATE_CUR_VALUE ("…/output/frame_rate; valtype=uint32")

7.2.1.3 Audio-related Capabilities

For an MIO component that supports audio media data types, it is required to support querying of following capabilities:

Query	Purpose
AUDIO_OUTPUT_SAMPLING_RATE_CUR_QUERY ("…/output/sampling_rate;attr=cur")	PVAuthor datapath uses this query to ask MIO of the sampling-rate (in Hz) of the capture device. The value is a 32 bit integer. The key returned from MIO would be: AUDIO_OUTPUT_SAMPLING_RATE_CUR_VALUE ("…/output/sampling_rate;valtype=uint32)
AUDIO_OUTPUT_NUM_CHANNELS_CUR_QUERY ("…/num_channels;attr=cur")	PVAuthor datapath uses this query to ask MIO of number of channels of the audio stream, capture device is sending. The value provided is a 32 bit integer. The key returned from MIO would be: AUDIO_OUTPUT_NUM_CHANNELS_CUR_VALUE ("…/num_channels;valtype=uint32")

7.2.2 releaseParameters

The releaseParameters API goes in pair with getParametersSync function to release the data that is returned by MIO in the previous getParametersSync call. This API is called when the PV module that requested the information, provided in the previous getParametersSync call, has finished using the data. If memory is allocated in the corresponding getParametersSync call, it can be deallocated or reused.

7.2.3 setParametersSync

The setParametersSync method is the key API of PvmiCapabilityAndConfig interface in today's usage. Other PV modules call this API to set various settings that are required typically to initialize the MIO itself as well as the hardware encoder or capture hardware.
-

7.2.4 verifyParametersSync

This API is called by PVAuthor datapath to verify whether the capability specified is supported by the MIO component and the hardware. The query strings listed in section 7.2.1 are used to identify the configuration to be verified. This API should return PVMFSuccess if the capability and value specified are supported or PVMFErrNotSupported if they're not supported.

8 Media Transfer

8.1 Media Transfer in PVPlayer

The MIO component receives media data from PVPlayer datapath, repackages the data into data structures that can be understood by decoder or renderer, and sends it downstream for further processing and rendering.

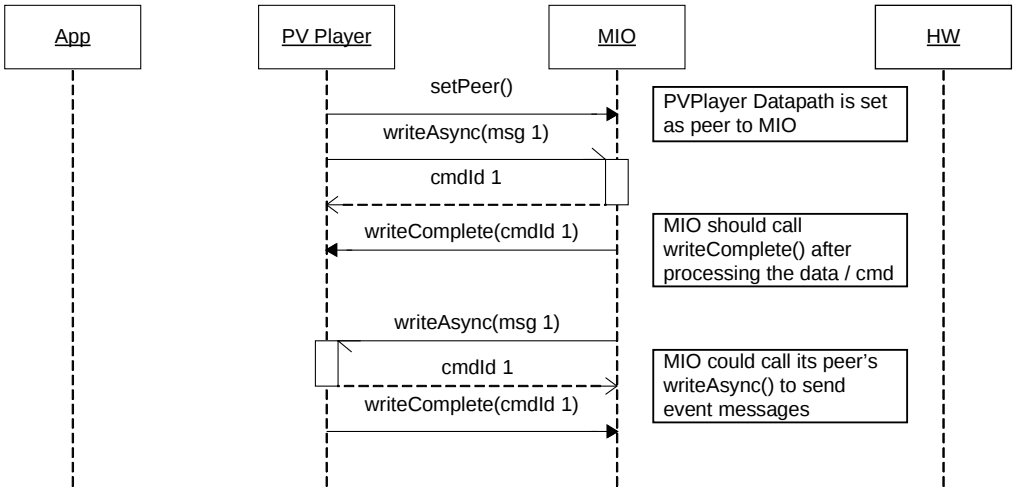

Figure 18: Sequence showing media transfer setup between PVPlayer and the MIO component.

8.1.1 setPeer

setPeer is an API inherited from PvmiMediaTransfer interface. This is the API that connects two peer modules that would be exchanging media data. The MIO component should store the peer pointer provided, and use that to communicate with the peer module.

8.1.2 writeAsync

The writeAsync API is inherited from PvmiMediaTransfer interface. From MIO component's point of view, the API's function is to receive media data and commands from PVPlayer datapath and process them asynchronously. This API is paired with writeComplete, and MIO component should call writeComplete on the peer module when it is done using the data. This would release the data provided in the writeAsync call back to PVPlayer datapath. The MIO component must process and consume the data in a FIFO order. Therefore it is generally necessary for MIO to maintain a queue of incoming writeAsync calls, and sequentially process them in an asynchronous fashion.

The MIO component would receive different types of media data and command, and they are identified by the value specified in the format_type and format_index parameters. The subsections below describe how the MIO component should handle the different types of data and command.

8.1.3 Format Specific Information

Format specific information such as the codec configuration data could be sent optionally from PVPlayer datapath through the writeAsync API. The data can be identified as format specific information if the following logic is true.

```
(format_type == PVMI_MEDIAXFER_FMT_TYPE_DATA) &&
(format_index == PVMI_MEDIAXFER_FMT_INDEX_FMT_SPECIFIC_INFO)
```

Since this information is already sent via setParametersSync(please refer to section 7.1.3), PVPlayer does not send this information via writeAsync any more.

8.1.4 Media Data

The MIO component can identify data exchanged through writeAsync API as media data if the following logic is true.

```
(format_type == PVMI_MEDIAXFER_FMT_TYPE_DATA) &&
(format_index == PVMI_MEDIAXFER_FMT_INDEX_DATA)
```

In the writeAsync function, in general it is only necessary to buffer up the data to a queue if it is not full, schedule the data to be processed asynchronously if the MIO and hardware are in the appropriate states, and return a unique command ID to the caller to identify the writeAsync call. This command ID should be used when writeComplete is called on the peer to complete the writeAsync call. If the MIO component is in a state that is ready to receive incoming data but the hardware is not, the MIO component should still accept and buffer up the data, and send it to hardware when the hardware enters the appropriate states.

The sequence diagram below describes generally how an MIO component should handle writeAsync calls, and calling writeComplete on the peer after the data is consumed.

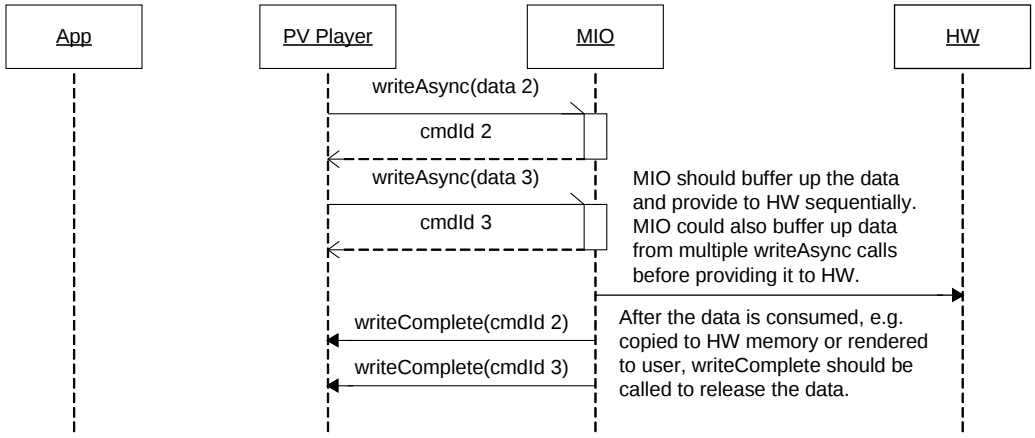

Figure 19: Media transfer sequence between PVPlayer and the MIO component.

Typically only active MIOs would queue multiple buffers before sending them to the hardware components. In some cases buffers may need to be combined, such as in the case of frame reconstruction for MIOs taking compressed input. In those cases care should be taken not to queue too many buffers without returning any or the datapath may run out of buffers to provide more data. This situation may arise for compressed data in a streaming scenario where an I-frame is broken into many separate buffers. The MIO component should copy data to an internal buffer and call writeComplete to return the buffer and avoid a potential deadlock situation.

8.1.4.1 Transfer Media Data to the Hardware

The transfer of media data to the hardware is specific to the target integration platform and can vary greatly from one platform to another. The MIO component is responsible for any necessary adaptation of the media data provided by the PVPlayer datapath before sending to the underlying components. It is expected that it should be possible to negotiate commonly understood data formats between the datapath and the MIO up-front and any modification of media data should be minor during steady-state processing.. While this adaptation is platform specific, the following are some examples of the functions that an MIO component might need to provide to adapt between the two layers.

- Request the hardware to allocate and provided access to shared memory for MIO component to write media data to.
- Store the media data and other related information to data structures that can be accepted by the hardware.
- For an active MIO component, it should manage the timing of providing media data to the hardware appropriately depending on the state of the datapath, the MIO component, and the hardware. For a passive MIO component, it should providing media data to the hardware as soon as possible.
- Manage the amount of data provided to the hardware by grouping a suitable number of media data buffers before providing it to decoder or renderer. This grouping logic may be needed to adapt between the different buffer sizes of the datapath and the renderer or to reconstruct a structural element such as a frame of data before passing it to a decoder in the case of an MIO taking compressed input.

8.1.4.2 Media Data Frame Reconstruction

In cases where the MIO component is acting as a renderer that takes compressed media data as input (i.e., the MIO is implementing the decoding and rendering directly), it is important to consider the streaming use-case where the data for a frame may be fragmented into multiple messages. If the underlying hardware components can accept these partial frames directly, then the MIO component does not need to do further processing and can pass the media data buffers without trying to reassemble larger blocks. The OpenMAX IL specification[6] requires components to be able to handle fragmentation of frames into separate buffers, so an MIO component interfacing to a compliant OpenMAX component should not need to do any frame reassembly. If the underlying hardware cannot handle partial frames, then frame reassembly inside the MIO component is required.

If media data frame reconstruction is required, the MIO component should use the `flags` field of the PvmiMediaXferHeader structure provided in `data_header_info` parameter of writeAsync to determine whether the data contained in writeAsync completes a media data frame or not. The `flags` field is a bitmask that contains a marker bit that is set to 1 if the writeAsync call contains the last data of a media data frame. The MIO component should use the following logic to check for frame boundary.

```
(data_header_info.flags & PVMF_MEDIA_DATA_MARKER_INFO_M_BIT)
```

If the above logic is true, it means the data is the last data of a media data frame. It is important to keep in mind that there may be packet loss so it may be necessary to also utilize the sequence number and timestamp information in the same structure to reliably detect missing data or a frame boundary. The datapath will deliver the media data ordered by sequence number so gaps indicate a data loss. The diagram below illustrates a frame reconstruction sequence based on the marker bit.

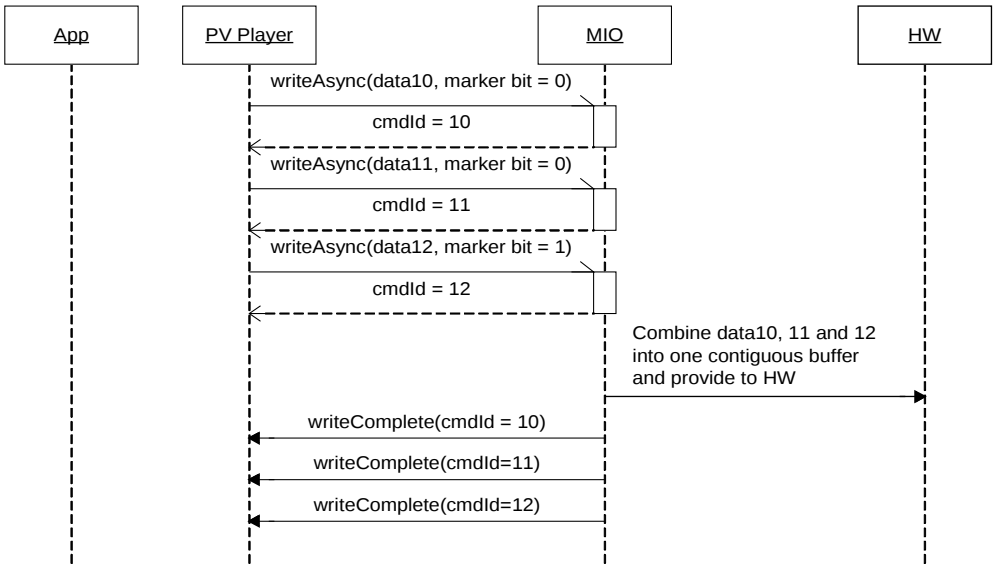

Figure 20: Sequence diagram showing use of the marker bit for frame reconstruction.

8.1.4.3 Flow Control

The processing of writeAsync function has an important role in flow control of media data in PVPlayer datapath. The MIO component and hardware should define a reasonable maximum size for the incoming data queue, and accept incoming data for asynchronous processing until the queue is full. When the MIO component or hardware data queue reaches its maximum size, the writeAsync call should leave by calling `OSCL_LEAVE(OsclErrBusy)`. This signals to PVPlayer datapath that MIO component cannot accept incoming data, and would then temporarily suspend transfer of media data.

When the MIO component can accept new incoming data again, for example, after the hardware has consumed some buffered data, MIO component should call `statusUpdate(PVMI_MEDIAXFER_STATUS_WRITE)` on the peer object to notify PVPlayer datapath that it can resume transfer of media data. The data that was rejected by the previous writeAsync call would be sent again, followed by subsequent media data.

This flow control method is only used for Media Data. Mio component should not do any leave for writeAsync of data notification like End of Data or Reconfig.

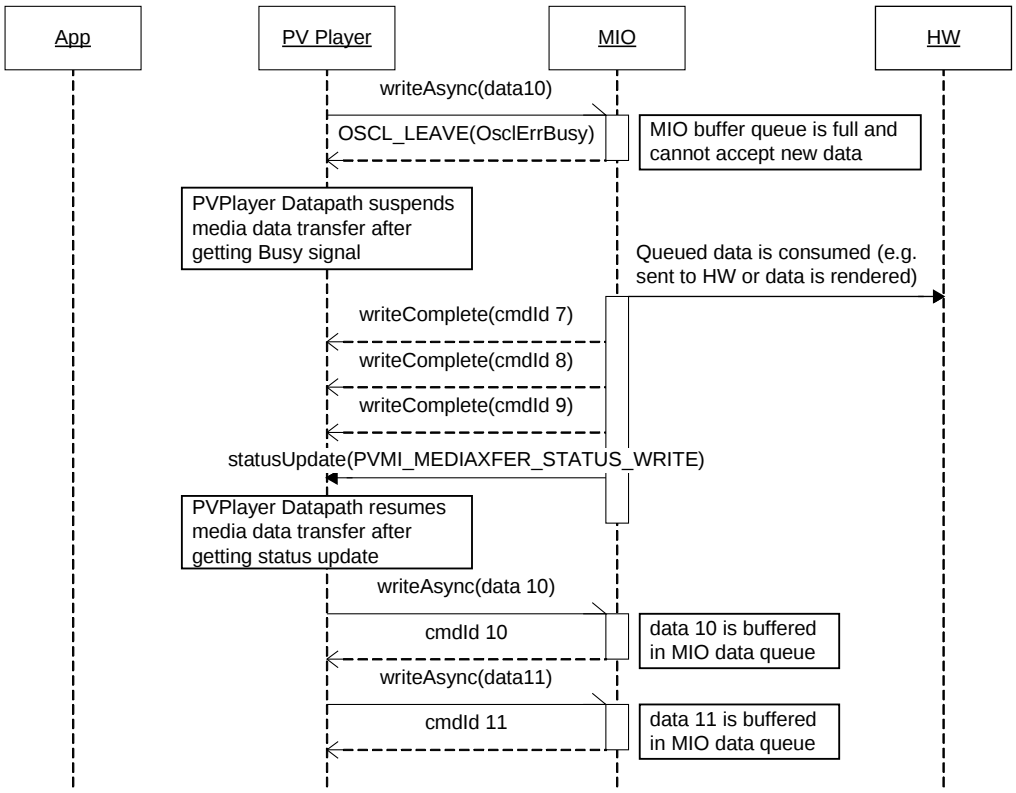

Figure 21: Sequence diagram showing the use of flow control of the media data.

8.1.4.4 Media Timestamp

The timestamp provided to MIO component is the presentation timestamp mapped from media sample timestamp in file format. This presentation timestamp is non-decreasing during a single playback session, i.e. the timestamp never goes backwards even if the playback repositions backwards. The purpose of such design in PVPlayer SDK architecture is because rendering is synchronized to a clock that does not go back in time. It would be impossible for the renderer to render media data at an accurate time if data timestamp or the clock could jump forward or backwards. The MIO component does not have access to the media sample timestamp from the media file format, and should not require this information to process the media data.

8.1.5 End of Data Notification

When playback reaches the end and there is no more media data to be sent to the MIO component, an end of data notification will be sent to the MIO component through writeAsync API. The MIO component can identify the end of data notification by the following logic

```
(format_type == PVMI_MEDIAXFER_FMT_TYPE_NOTIFICATION) &&
(format_index == PVMI_MEDIAXFER_FMT_INDEX_END_OF_STREAM)
```

When MIO component receives this notification, it should continue to send data to the hardware until all data is exhausted and inform the hardware that end of data is reached. The hardware should finish rendering all provided data and then stop playback. During this time, the MIO component wait until the hardware indicates that all data has been rendered before calling writeComplete to the peer for the corresponding writeAsync that provided the end of data notification. It is important to ensure that the timing of this writeComplete comes after all data has been rendered.

After the end of data notification is completed, PVPlayer Engine would automatically pause the datapath, and inform the application that end of data has been reached.

Mio component should not call `OSCL_LEAVE(OsclErrBusy)` here for end of data notification.

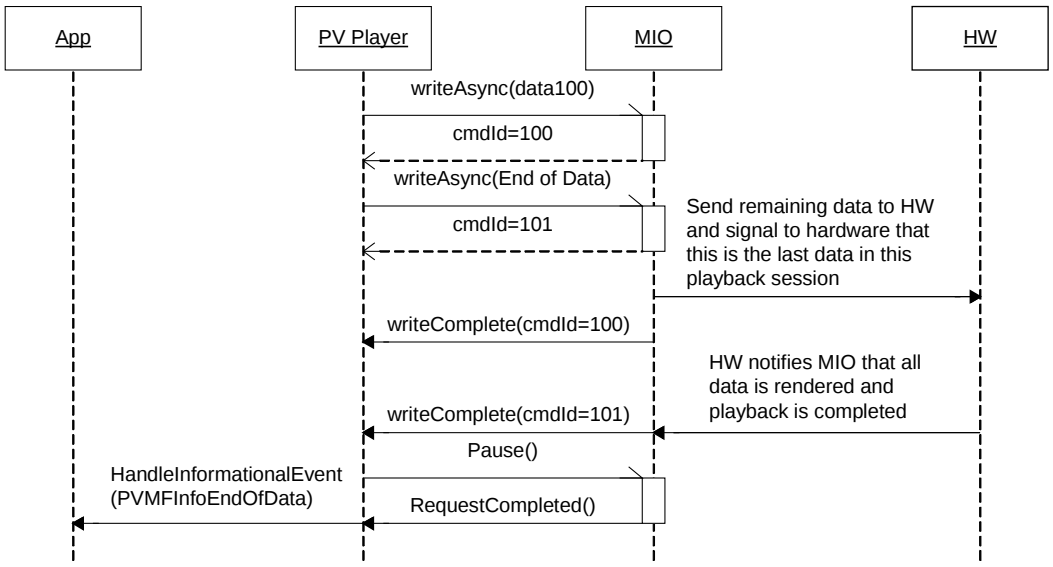

Figure 22: Sequence diagram showing end of data notification.

8.1.6 Reconfig Notification

When PVPlayer decided to do the stream changing, a reconfig notification will be sent to the MIO component through writeAsync API. A typical usage is when network condition turns bad, PVPlayer chooses lower bitrate stream instead of higher one. The MIO component can identify the reconfig notification by the following logic

```
(format_type == PVMI_MEDIAXFER_FMT_TYPE_NOTIFICATION) &&
(format_index == PVMI_MEDIAXFER_FMT_INDEX_RE_CONFIG_NOTIFICATION)
```

When MIO component receives this notification, it should do the following steps

- Continue to send data to the hardware until all data earlier than this notification is consumed.

- Wait until the hardware indicates that all data has been rendered.
- Do necessary steps to config the hardware to a state that ready to receive new data stream. The necessary steps are hardware dependent and may include Stop, Reset, Init, Start etc. Some hardware does not need any re-config before receiving new data stream. If the process to config the hardware is an async process, MIO component should ensure it could works fine even PVPlayer call pause() or start() of MIO component before the config process finished,
- Call writeComplete to indicate the re-config of the hardware is finished.
- Starting processing the new data stream.

Mio component should not call `OSCL_LEAVE(OsclErrBusy)` here for reconfig notification.

8.1.7 Event Reporting from MIO to PVPlayer datapath

To report unsolicited error or information events from MIO component to PVPlayer datapath, the MIO component should call the peer's writeAsync function. To indicate to the peer that the writeAsync contains event information, the MIO component should call writeAsync with the following parameters

Parameter	Value
`format_type`	`PVMI_MEDIAXFER_FMT_TYPE_NOTIFICATION`
`format_index`	`PVMI_MEDIAXFER_FMT_INDEX_ERROR_EVENT` for error events `PVMI_MEDIAXFER_FMT_INDEX_INFO_EVENT` for info events
`data`	Pointer to an `PVMFAsyncEvent` object. If extended errors are defined for the MIO component, a `PVMFBasicErrorInfoMessage` object should be created to contain the extended error event. The appropriate constructor of `PVMFAsyncEvent` should be used to pass the `PVMFBasicErrorInfoMessage` object as `aEventExtInterface` parameter.
`data_len`	Size of `PVMFAsyncEvent` object

PVPlayer datapath would call the MIO component's writeComplete function after the event is processed. If the event data is allocated dynamically, it should be deallocated upon writeComplete is received.

Not all the events are sent to application. Only the events with following EventType (a member of `PVMFAsyncEvent`) are sent to application:

For PVMI_MEDIAXFER_FMT_INDEX_ERROR_EVENT

PVMFErrCorrupt	PVMFErrOverflow	PVMFErrUnderflow
PVMFErrTimeout	PVMFErrNoResources	PVMFErrResourceConfiguration
PVMFErrResource	PVMFErrNoMemory	PVMFErrProcessing,

For PVMI_MEDIAXFER_FMT_INDEX_INFO_EVENT for info events

PVMFInfoDataDiscarded	PVMFErrNoResources	PVMFSuccess

The sequence diagram in Figure 23 below illustrates how an error from hardware can propagate up to the application level.

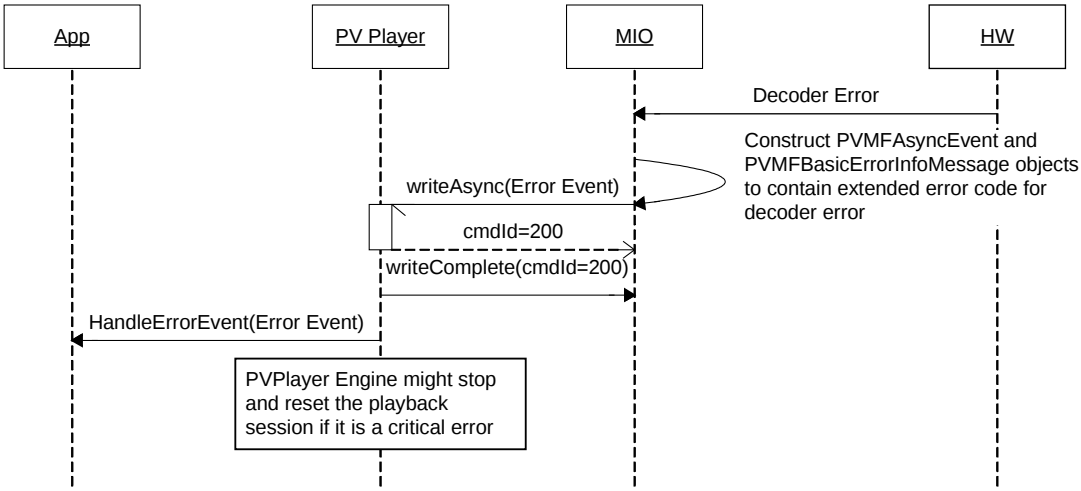

Figure 23: Sequence diagram showing error event propagation.

8.1.8 writeComplete

The implementation of `writeComplete` for MIO component should be simple because it would only be called to complete a previous `writeAsync` issued by MIO component to send events to PVPlayer datapath. The `writeComplete` function of MIO component should deallocate any memory that was allocated for the `writeAsync` it is completing. This might include the `PVMFAsyncEvent` and `PVMFBasicErrorInfoMessage` objects that were used in reporting the event depending on implementation.

8.1.9 Unsupported APIs

There are several APIs defined in `PvmiMediaTransfer` interface that MIO component for PVPlayer doesn't need to support

- `UseMemoryAllocators`
- `ReadAsync`
- `ReadComplete`

For these APIs, the MIO component should call `OSCL_LEAVE(OsclErrNotSupported)` to indicate they are not supported.

8.2 Media Transfer in PVAuthor

PVAuthor datapath sets itself as a peer to the MIO component. MIO gets the data from Hardware Capture device and sends it back to the PVAuthor datapath by calling the WriteAsync method of its peer.

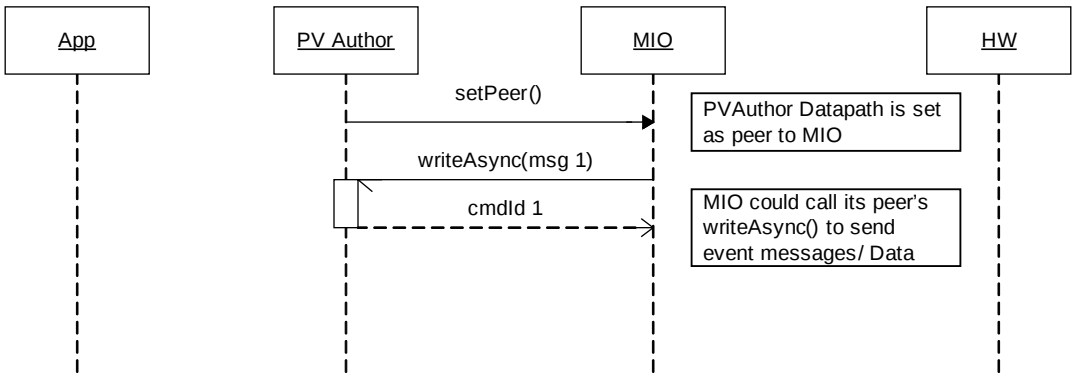

Figure 24: Sequence showing media transfer setup between PVAuthor and the MIO component.

8.2.1 SetPeer

SetPeer is an API inherited from PvmiMediaTransfer interface. This is the API that connects two peer modules that would be exchanging media data. The MIO component should store the peer pointer provided, and use that to communicate with the peer module. The peer of MIO component is the media input node.

8.2.2 WriteAsync call on MIONode

WriteAsync is an API inherited from PvmiMediaTransfer interface. From the MIO component's point of view, the method is used to send the media data and commands to PVAuthor datapath and process them asynchronously. This API is paired with writeComplete, and media input node should call writeComplete on the MIO component when it is done using the data. The writeAsync call returns an ID which can be saved in a queue. Once writeComplete for the corresponding ID is returned (after writeAsync is processed), the saved ID can be deleted and the corresponding buffer/data pointer can be deallocated or recycled. The MIO component must provide data in the order it should be processed downstream.

The MIO component would send different types of media data and command, and they are identified by the value specified in the format_type and format_index parameters. The subsections below describe how the MIO component should handle the different types of data and command.

8.2.3 Format-Specific Information

Format specific information such as codec header, formats and encoders supported by the capture device could be sent optionally to the PVAuthor datapath through the writeAsync API. The data can be set as format specific information by making following assignments:

```
format_type = PVMIPVMI_MEDIAXFER_FMT_TYPE_NOTIFICATION
format_index = PVMI_MEDIAXFER_FMT_INDEX_FMT_SPECIFIC_INFO
```

Typically the format specific information is already provided to MIO component prior to writeAsync calls through capability configuration exchange. Therefore the MIO component can call writeComplete to the peer to release this data without parsing the information here unless the hardware requires the format specific data. In case that processing of format specific information is again necessary, the MIO should process the information similar to how it is done in setParametersSync, and call writeComplete to the peer to release the data after it is consumed. Please refer to section 7.2.3 and its subsections for more details

8.2.4 Media Data

The MIO component can send media data through the writeAsync API by setting the following values in format type and format index.

```
format_type = PVMI_MEDIAXFER_FMT_TYPE_DATA
format_index = PVMI_MEDIAXFER_FMT_INDEX_DATA
```

The MIO component receives the captured data from the source and sends this received data to the PVAuthor engine datapath by calling the writeAsync function of its peer. The sequence diagram below illustrates how an MIO component should call writeAsync, and how it should handle the writeComplete from the peer after the data is consumed.

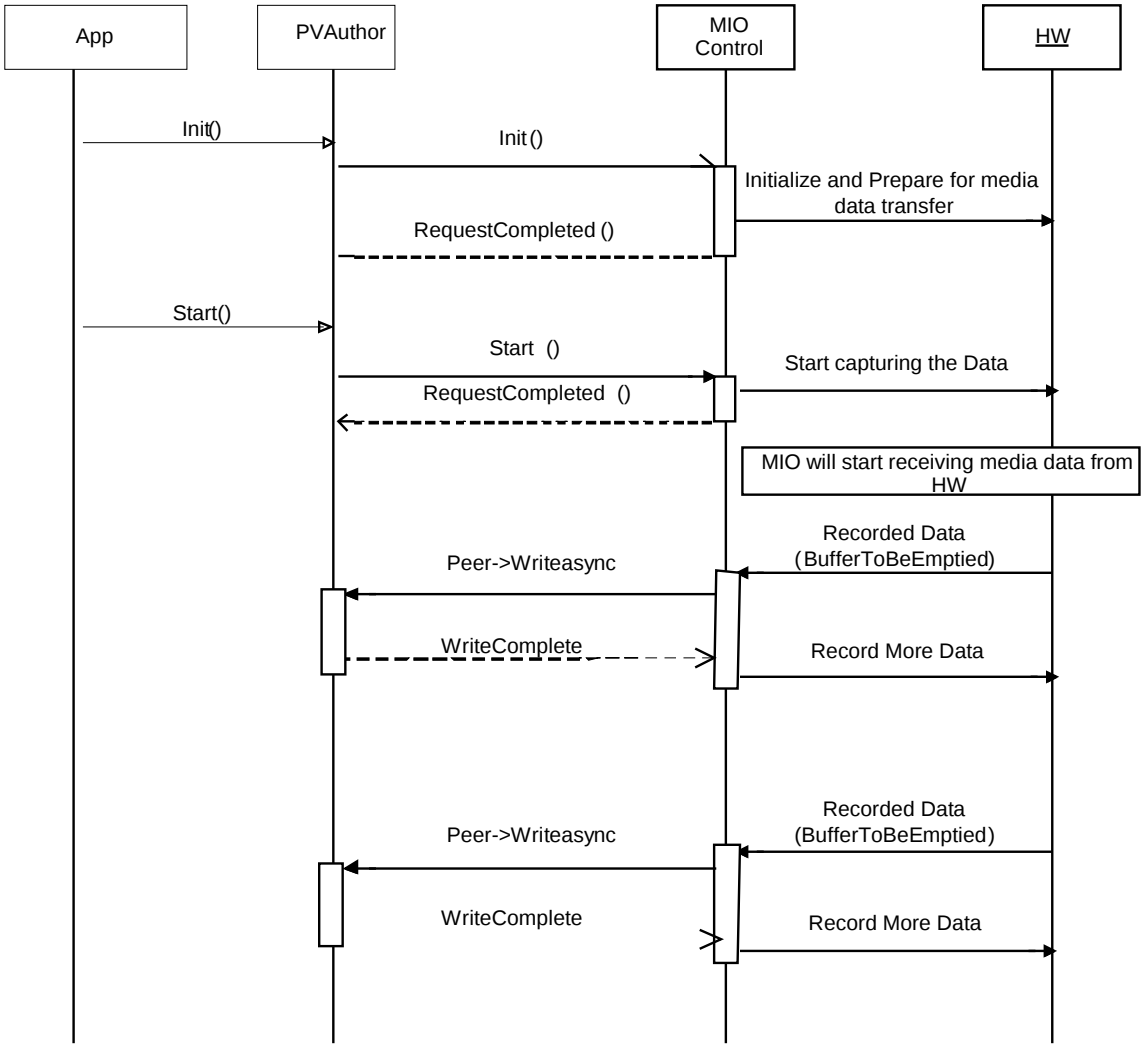

Figure 25: Media transfer sequence between PVAuthor and the MIO component.

8.2.4.1 Capturing Media Data

The transfer of media data from the hardware to MIO is specific to the target integration platform and can vary greatly from one platform to another. The MIO component implementation needs to adapt to these differences in hardware interfaces. The implementation needs to handle how the hardware provides media data to MIO (e.g., it could be in form of callbacks to MIO). Then MIO needs to format the data received in a way so that it can be sent to PVAuthor datapath. While this adaptation is platform specific, the following are some examples of the functions that an MIO component might need to provide to adapt between the two layers.

- Request the hardware to allocate and provided access to shared memory for MIO component to read media data from.
- Store the media data and other related information received from hardware interface to data structures that can be accepted by the PVAuthor datapath.

In the sequence diagram above, taking Symbian OS implementation as an example, Symbian OS hardware interface is Devsound. MIO derives from Devsound observer. Devsound communicates with MIO using callback functions in this observer. Devsound supplies data to the MIO using a handshake mechanism. Data captured by the Devsound is passed to the MIO in a buffer in the observer callback BufferToBeEmptied. MIO copies the data from the buffer supplied and passes on to the PVAuthor datapath by calling writeAsync method of the peer. MIO then asks for more data from Devsound by calling its RecordData function. This handshake mechanism goes on till authoring process is paused or stopped or some error occurs.

8.2.4.2 Flow Control

The processing of writeAsync function has an important role in flow control of media data in PVAuthor datapath. The MIO component has to deal with situations when the incoming data queue of the PVAuthor datapath may be unable to accept any more data because it is already full. When the PVAuthor datapath data queue accepting data from MIO component reaches its maximum size, the writeAsync call should leave by calling `OSCL_LEAVE(OsclErrBusy)`. This signals to MIO component that PVAuthor datapath cannot accept incoming data, and would then temporarily suspend transfer of media data. MIO component then should delay the callback to hardware which informs the Hardware to capture more data till the time PVAuthor datapath can accept more data.

After this there are two ways to find out about the availability of more buffers at the PVAuthor datapath. One is to try to send more data to PVAuthor datapath every time the PVAuthor datapath calls writeComplete call back in the MIO component. This way writeAsync call would succeed when PVAuthor datapath can accept more incoming data.

In the second method, When the PVAuthor datapath can accept new incoming data again, for example, after PVAuthor has consumed some buffered data, PVAuthor datapath will call `statusUpdate(PVMI_MEDIAXFER_STATUS_WRITE)` on the MIO component to notify that it can resume transfer of media data. The data that was rejected by the previous writeAsync call needs to be sent again, followed by subsequent media data.

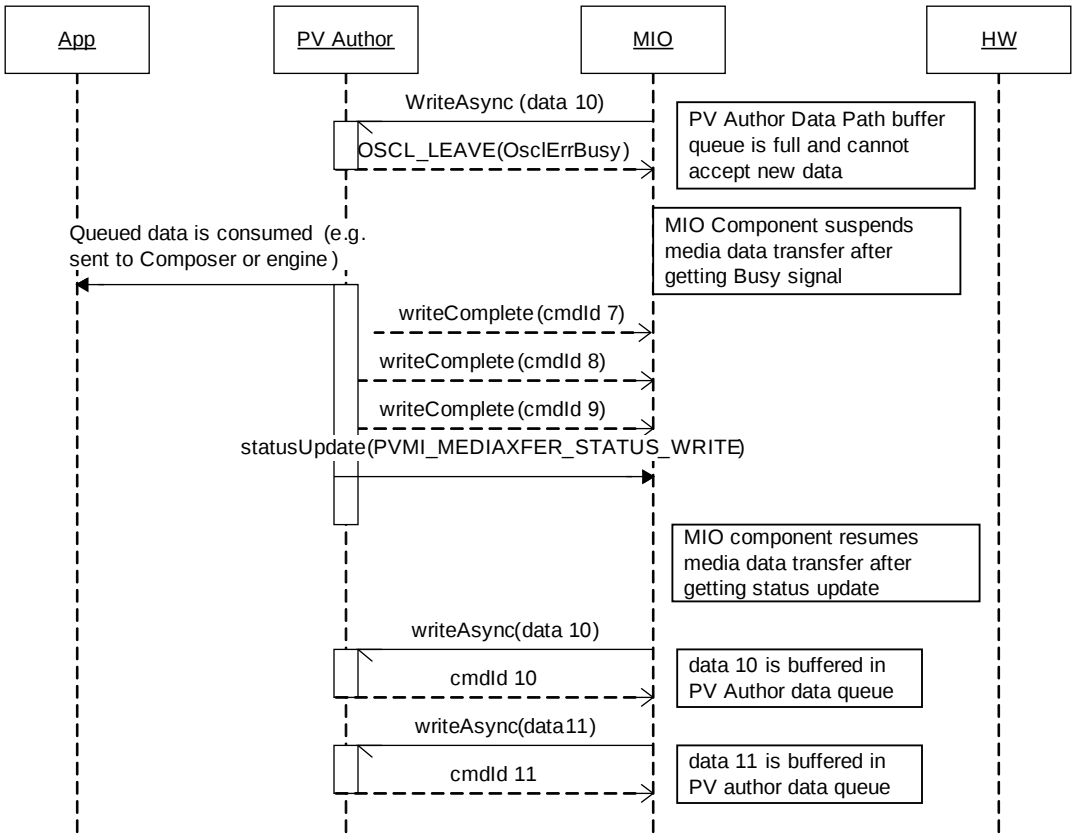

Figure 26: Illustration of media data flow control between PVAuthor and the MIO component.

8.2.4.3 Media Timestamp

The capturing device provides the timestamp associated with the captured data to the input MIO component. MIO should send these timestamps back to the PVAuthor datapath by updating the timestamp variable of the PvmiMediaXferHeader while sending the data back in writeAsync calls to the PVAuthor datapath. The gaps induced in the timestamp due to reasons like error in capturing or non availability of buffers to fill the data at the PVAuthor datapath needs to be taken care of by the MIO component.

8.2.4.4 Media Data Frame Reconstruction

It is possible that some hardware platforms support partial frame processing and that they can give partial frames to MIO. For such hardware that support processing of partial frame media data, the MIO component would be required to inform the PVAuthor datapath about the partial frames.

If media data frame reconstruction is required, the MIO component should use the `flags` field of the PvmiMediaXferHeader structure provided in `data_header_info` parameter of writeAsync to indicate that the data contained in writeAsync completes a media data frame or not. The `flags` field is a bitmask that contains a marker bit that is set to 1 if the writeAsync call contains the last data of a media data frame. The PVAuthor datapath should use the following logic to check for frame boundary.

```
(data_header_info.flags & PVMF_MEDIA_DATA_MARKER_INFO_M_BIT)
```

If the above logic is true, it means the data is the last data of a media data frame.

8.2.5 writeComplete of MIO Component

The implementation of `writeComplete` for MIO component should be simple because it would only be called to complete a previous `writeAsync` issued by MIO component to send data to PVAuthor datapath. The `writeComplete` function of MIO component should deallocate any memory that was allocated for the `writeAsync` it is completing. This might include the `PVMFAsyncEvent` and `PVMFBasicErrorInfoMessage` objects that were used in reporting the event depending on implementation.

8.2.6 UseMemoryAllocators

This API has been deprecated.

8.2.7 Unsupported APIs

There are several APIs defined in `PvmiMediaTransfer` interface that MIO component for PVAuthor doesn't need to support

- `ReadAsync`
- `ReadComplete`
- `writeAsync`

For these APIs, the MIO component should call `OSCL_LEAVE(OsclErrNotSupported)` to indicate they are not supported.

9 Temporal Synchronization for Playback

In Section 6 of *PVPlayer Developer's Guide* document[6], the PVPlayer SDK is required to render all the multimedia data that it handles in a temporally synchronized manner also known as "AV sync". To achieve AV sync, the data sink in PVPlayer datapath is responsible for synchronization of media rendering with the playback clock, and the synchronization of the presentation of different media tracks. PVPlayer Engine owns and manages the playback clock to report playback progress to the application, and the data sink of each media track uses it to synchronize rendering of media data. For more information about playback clock and other details about PVPlayer SDK architecture for temporal synchronization, please refer to Section 6 of *PVPlayer Developer's Guide* document[6].

The MIO component, being a data sink component, can optionally assume the responsibility to synchronize rendering. And an MIO component that assumes such responsibility is categorized as an active data sink. If the MIO component is implemented as an active data sink, it should implement the `PvmiClockExtensionInterface` and `OsclClockStateObserver` interfaces. PVPlayer datapath would recognize through the support of these extensions that the MIO component is an active data sink, and would defer the responsibility to synchronize rendering to the MIO component. Typically the MIO component should behave as active data sink if it is going to render compressed data, as underlying decoders will take some finite time in decoding and rendering. To achieve better AV-sync, MIO component should behave as active data sink even it is going to render uncompressed data.

Conversely, an MIO component can defer this responsibility to the data sink node that is a part of PVPlayer datapath. Such MIO component is categorized as a passive MIO data sink. If the MIO component is implemented as a passive data sink, it does not need to implement `PvmiClockExtensionInterface` and `OsclClockStateObserver` interfaces. PVPlayer datapath would recognize that the MIO component is a passive data sink, and would be responsible for sending data to MIO component at the appropriate time.

This section focuses on describing how to implement an active MIO component that is responsible for synchronization of rendering.

9.1 Role of PVPlayer SDK Modules in Synchronization

Before examining how to implement an active MIO component, the role of PVPlayer related modules in synchronization should be explained first. As mentioned in the above introduction for synchronization, PVPlayer Engine owns and manages the playback clock, which it uses to report playback progress to the application, and the data sink of each media track uses it to synchronize rendering of media data. And for an active MIO component, the synchronization is done by the MIO component. The class diagram below illustrates the relationship between various PVPlayer modules for synchronization

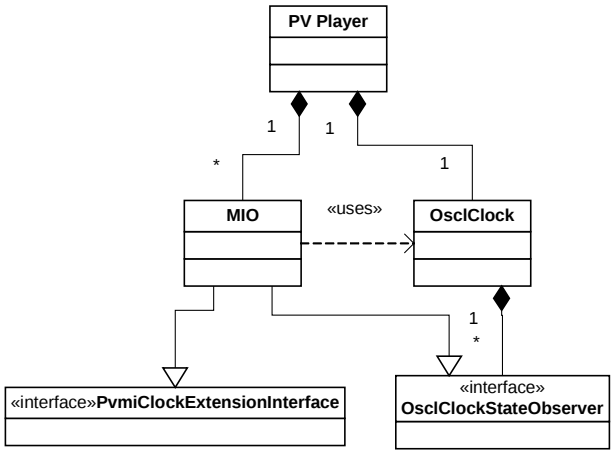

Figure 27: Class diagram synchronization-related classes within PVPlayer.

9.1.1 Providing Media Clock to MIO

PVPlayer queries the MIO component for an instance of PvmiClockExtensionInterface, and an active MIO component should respond with its implementation of PvmiClockExtensionInterface. PVPlayer would then provide a pointer to the playback clock to MIO component through `PvmiClockExtensionInterface::SetClock` API. The MIO component should store the pointer, as it will be used throughout the playback. The MIO component can use the various functions in OsclClock class to control, adjust and read the playback clock for synchronization. All MIO components in the datapath are provided with the same playback clock object to facilitate synchronization of multiple media tracks. The sequence diagram below illustrates how playback clock is provided to MIO component.

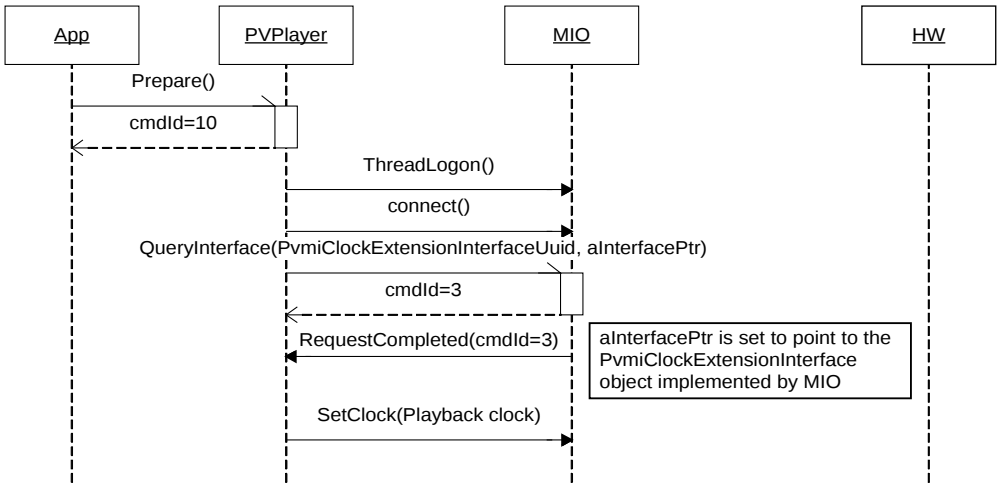

Figure 28: Initialization of the clock reference for active MIO components.

9.1.2 Providing Playback Progress to Application

PVPlayer Engine supports periodic reporting of playback progress to application. PVPlayer Engine periodically retrieves the playback clock value and calculates the playback position based on the value. The playback position is then reported to the application in PVMFInfoPositionStatus events. The application is expected to use the position reported by PVPlayer to display playback time or progress bar to the user. Therefore, it is important that the playback clock is accurately synchronized with the actual playback progress, such that accurate information is presented to users. For details, please refer to Section 11 of *PVPlayer Developer's Guide* document[6].

9.2 Audio Synchronization

Audio rendering typically does not need to use an external clock object to synchronize rendering because the audio device is configured to consume audio data at the sampling rate of incoming data. Therefore, the rate of audio data consumption should be basically the same as the playback clock rate. However, for PVPlayer to report accurate playback progress to user, some logic would required to control and update the playback clock such that it accurately matches with audio rendering progress.

9.2.1 Synchronize With the Start of Audio Rendering

In Section 18, it was mentioned that during processing of PVPlayerInterface::Start command, OsclClockStateObserver::ClockStateUpdated() would be called to notify start of the playback clock. This start of playback clock signals the start of media rendering, and the MIO component

should request the hardware to start rendering as soon as possible. However, it is also typical that the hardware needs additional time to start rendering, or requires more media data to be buffered before it can start rendering. In such case, the start of playback clock is not matched with the actual start of audio rendering, causing an offset in playback progress reported to the application by PVPlayer and the actual playback progress. Therefore, for most MIO components, it is necessary for it to pause the playback clock immediately upon getting notified that the clock is started, and in parallel continue to buffer and request the hardware to start rendering. The hardware should report to MIO component when the first media data is rendered to user, and the MIO component should start the playback clock again upon such event. This would ensure the playback clock is started at the same time as the start of rendering, and eliminate any possible offset between playback progress reported to application and actual rendering progress.

The function OsclClockStateObserver::ClockStateUpdated() could be called anytime clock state is changed, even this change is done by MIO itself. The MIO should keep track whether the clock state change is due to its own clock commands or those from the application/pvPlayer calling pause/resume/stop on the clock.

The sequence diagram below illustrates one method to synchronize the start of audio rendering with the start of playback clock in case there is some initial latency.

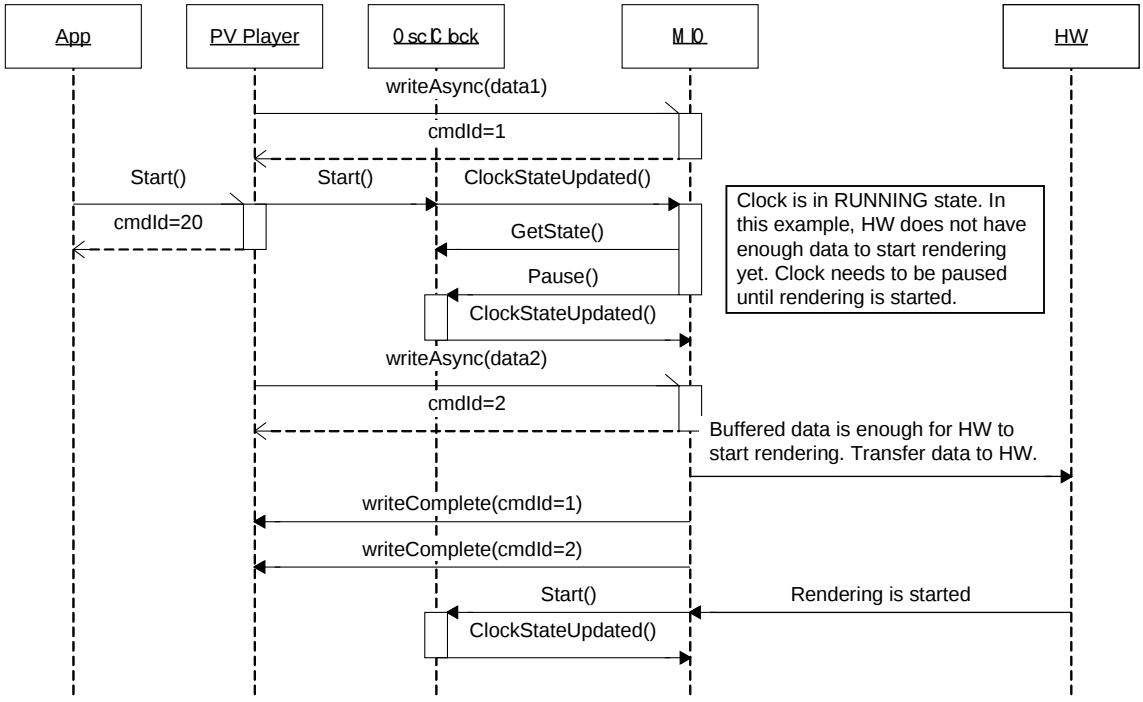

Figure 29: One method of handling initial rendering latency in an active MIO component.

For an active MIO component, the PVPlayer datapath continues sending data to the MIO component when clock state is PAUSED or RUNNING, and MIO component should continue to buffer the received data and decide when to send data to the hardware.

For a passive MIO component, the PVPlayer datapath sends data to the MIO component only when clock state is RUNNING, and MIO component should send these data to the hardware as soon as possible.

Please note that all the operations with OsclClock like Pause(), Start() from the MIO component must be done within the same thread context as the PVPlayer, where the clock was created

9.2.2 Synchronize After Repositioning

In Section 4.4, it was mentioned that when the application requests for repositioning, buffered data in MIO component and hardware would be discarded and playback would start from the new position. Similar to the start of playback described in Section 9.2.1, it is also necessary for an audio MIO component to control the playback clock such that it is synchronized with the start of rendering of new data.

Function OsclClockStateObserver::ClockStateUpdated() could be called anytime clock state is changed, even this change is done by MIO itself. MIO should judge that the clock state change is due to application/pvPlayer call pause/resume/stop or MIO pause/start the clock.

The sequence diagram illustrates how to synchronize the start of rendering of audio data from new position with the start of playback clock.

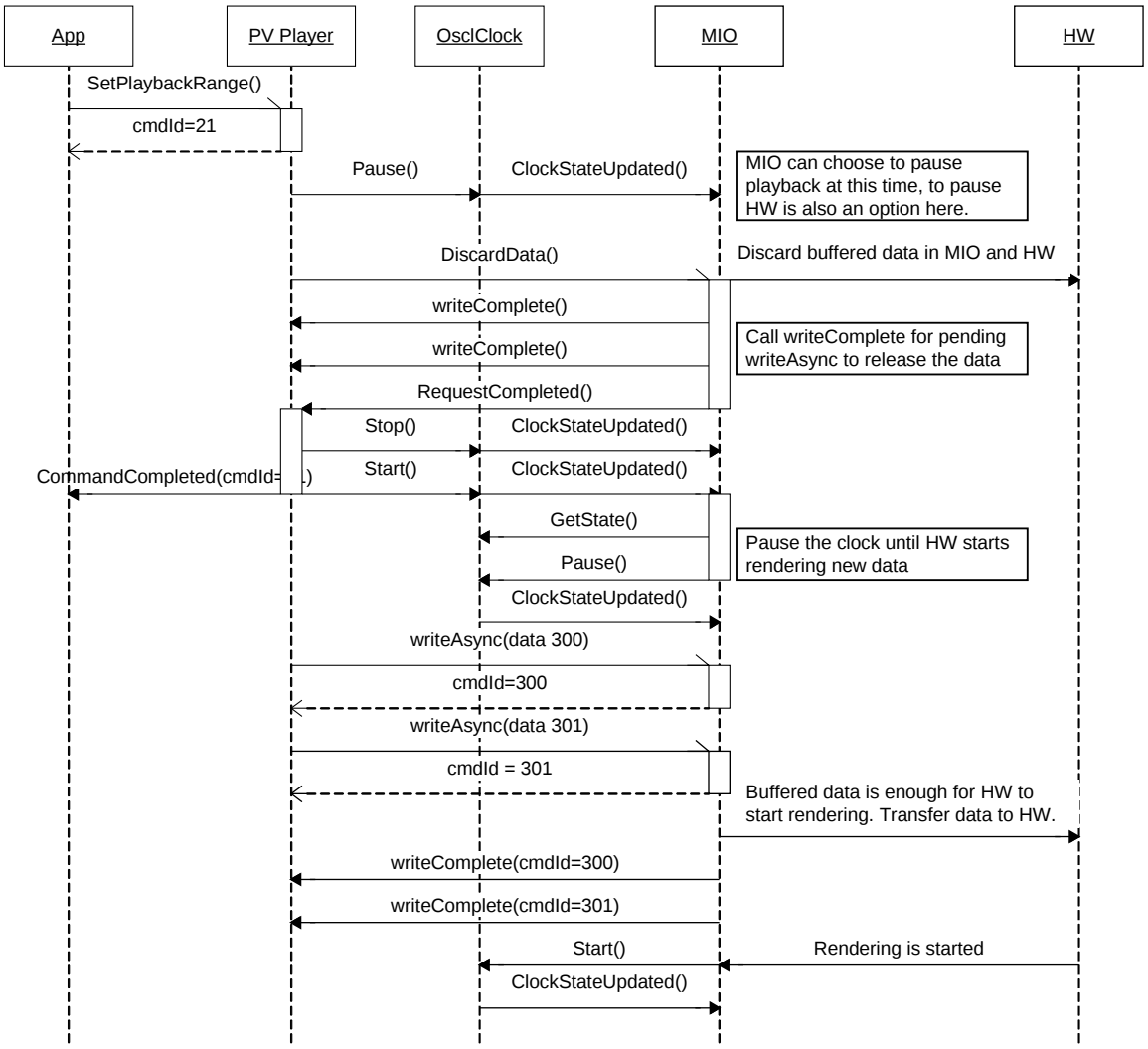

Figure 30: Synchronization after repositioning.

9.2.3 Synchronize During Playback

Although the rate of audio data consumption should be basically the same as the playback clock rate, they are still running independently from each other, and hence there are inevitably some differences between them. In general, the difference between playback clock rate and audio data consumption rate is very small. However, this difference would accumulate over time, and if the playback session lasts for a long time, the accumulated difference could become noticeable by users. Therefore, it is necessary for the hardware to report to MIO component the audio rendering progress, and MIO component should use it to adjust the playback clock such that the playback clock is synchronized with audio rendering progress. The audio rendering progress can

be reported actively by the hardware or MIO component can passively query the hardware for the progress. The playback progress can be reported in terms of time or in number of samples rendered, which the MIO component can convert into time value for comparison with playback clock value.

As mentioned earlier, the difference between playback clock and audio rendering progress should be minor over a short period of time as long as the start of playback clock is synchronized with start of audio rendering. Therefore, this adjustment of playback clock should be done occasionally and only if the delta between playback clock and audio rendering progress is larger than a suitable threshold.

The sequence diagram below illustrates an example of playback clock adjustment.

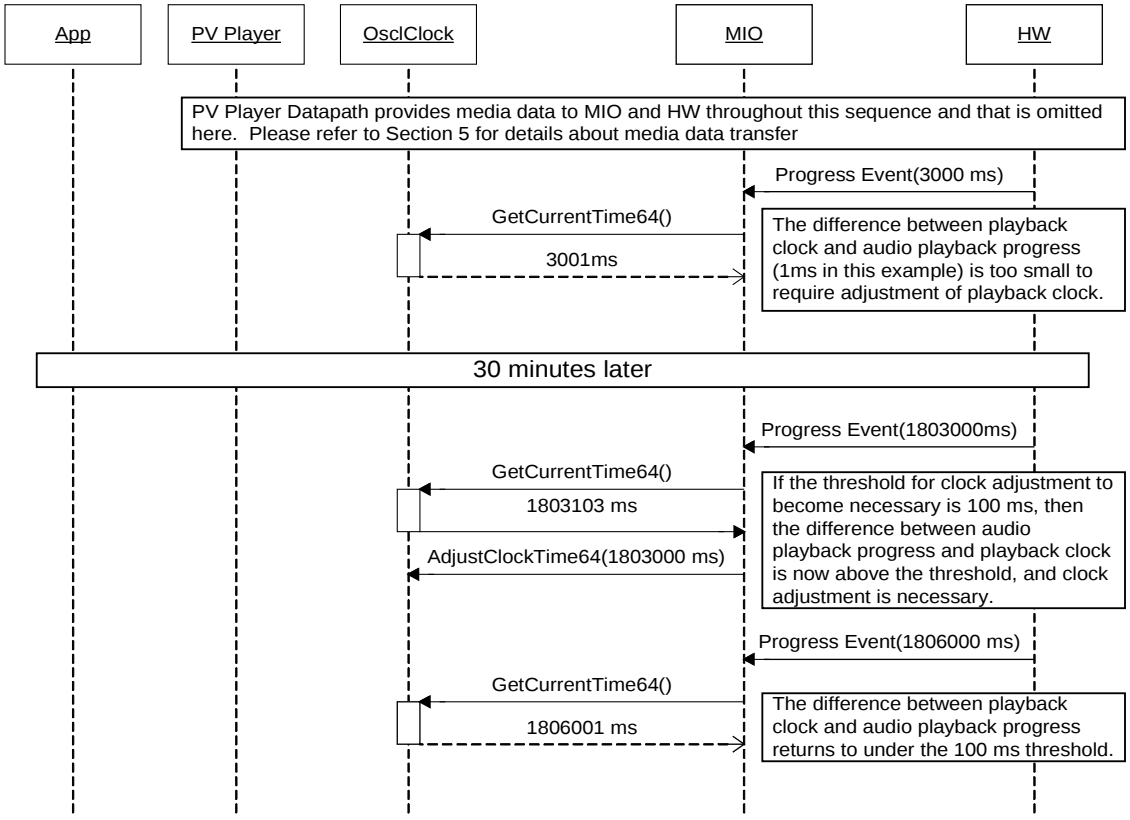

Figure 31: Sequence diagram showing clock adjustments during playback to account for drift.

9.3 Video Synchronization

Unlike audio rendering, video rendering requires reference to a clock to determine the appropriate time to render a particular video frame. The video renderer should schedule rendering of a video frame at the closest possible clock time to the video frame timestamp. As discussed in Sections

9.1 and 9.2, PVPlayer maintains a playback clock object that is synchronized with audio playback. Therefore, if the video renderer synchronizes video rendering with the same playback clock, then the video rendering would be properly aligned with audio rendering, presenting a desired experience to the user.

9.3.1 Video Rendering Without Hardware Assistance

On some platforms, the MIO component receives uncompressed YUV or RGB video data, and routes the video data to the application for rendering without hardware assistance. It is possible for such MIO component to be passive and defer to PVPlayer datapath to perform video synchronization. However, if the MIO component is implemented as an active component, it should schedule the transfer of video data to the application according to the playback clock. The sequence below illustrates how the MIO component can synchronize video rendering.

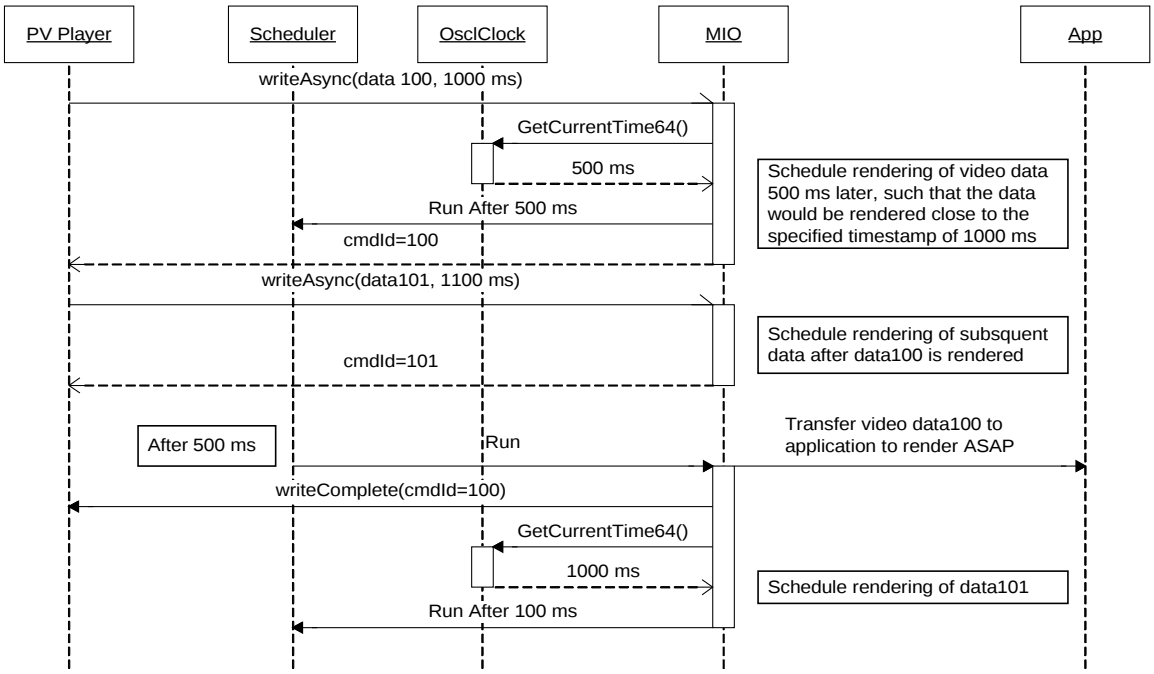

Figure 32: Sequence diagram of video rendering scheduling.

9.3.2 Video Rendering With Hardware Assistance

If a hardware decoder or renderer is used, the hardware must support the necessary APIs for the MIO component to provide the playback clock object to it. The hardware must use the playback clock provided for synchronization of video rendering. In this case, the MIO component only

needs to push incoming video data down to the hardware as soon as possible and defer to the hardware to schedule rendering according to the media clock.

9.4 Audio-Video Synchronization

Audio-video synchronization is the end goal of implementing audio and video synchronization mechanism described in Sections 9.2 and 9.3. In PVPlayer SDK architecture, the media clock is adjusted to match with audio rendering progress. On the video rendering side, the video renderer should schedule video frames to be rendered according to when frame timestamp is close the value of the media clock. This combination of interactions between playback clock, audio MIO component and video MIO component results in audio-video synchronization. Figure 33 Illustrates the interactions between the modules that achieves audio-video synchronization.

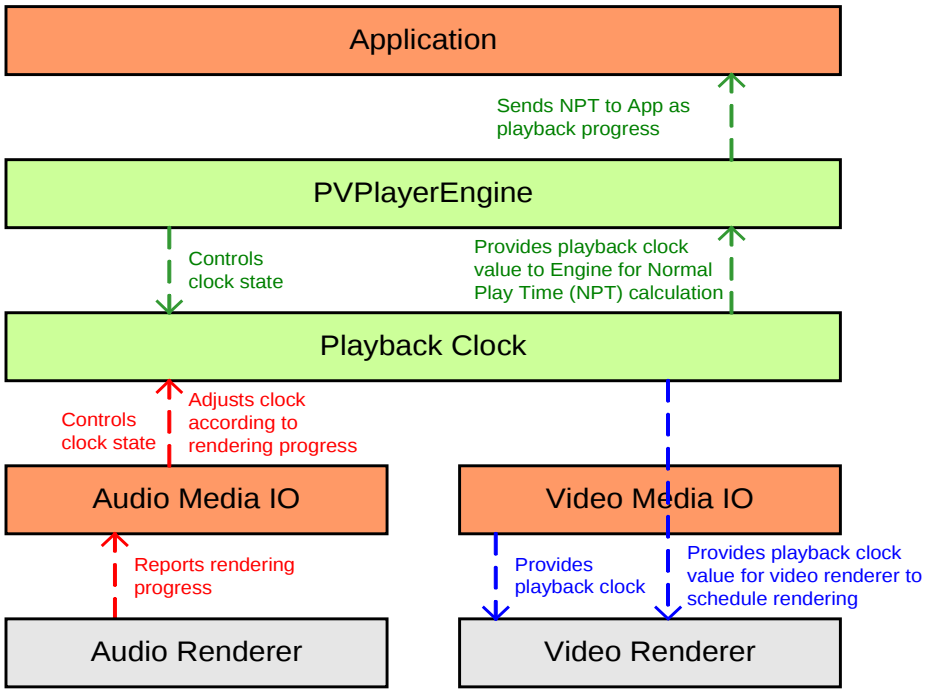

Figure 33: A diagram of the interactions involved in A/V synchronization.

10 Appendix

10.1 Media clock facts and properties.

- The media clock is a monotonic non-decreasing clock that will never go backwards, even when random positioning within a media stream (clip). It is used to make “local”

decisions, independent of the absolute position in the media stream, about whether it is time to render specific media samples.

- The instance of the media clock used for rendering (created by pvplayer) does not represent the actual position in the clip or normal play time (NPT). Pvplayer maintains a mapping between NPT and the media clock to report NPT to the UI.
- A reference to the common media clock is passed to all MIO components that implement the clock extension interface.
- Adjustment can be done only in RUNNING State.
- OsclClockStateObserver:: ClockStateUpdated() call will be made if state is changed.
- Any changes to the clock value or clock state must be made within the same thread context as the player/author engine is running (i.e., the same thread where the clock was created).

10.2 Video dimensions passed to MIO comp

PV Engine extracts video dimension for various bit stream as follows.

For H263 codecs
Display_Dimension: - It extracts the display information (display width and display height) from h263sampleentry atom. If these values are equal to 0, then we calculate these values from the decoder.
Decode_Dimension: - It extracts the decode dimension always from the decoder which is always a multiple of 16.

For H264 and M4V codecs
Display_Dimension: - It extracts the display information from decoder. These values do not necessarily be a multiple of 16.
Decode_Dimension: - It extracts the decode dimension always from the decoder which is always a multiple of 16.
This information is extracted by the source node and provided to various modules in this way

To Video MIO Comp
These two sets of dimensions [Display Dimension and Decode Dimension] are passed from PVEngine to Video MIO using setParametersSync api during capability exchange

Decode_Dimension
```
MOUT_VIDEO_WIDTH_KEY "x-pvmf/video/render/width;valtype=uint32"
MOUT_VIDEO_HEIGHT_KEY "x-pvmf/video/render/height;valtype=uint32"
```

Display_Dimension
```
MOUT_VIDEO_DISPLAY_WIDTH_KEY
"x-pvmf/video/render/display_width;valtype=uint32"
MOUT_VIDEO_DISPLAY_HEIGHT_KEY
"x-pvmf/video/render/display_height;valtype=uint32"
```

In case of H263 it will come from h263sampleentry atom if not 0,otherwise from the decoder.