# PVPlayer SDK Developer's Guide
# OHA 2.07, rev 2
# Jun 25, 2010



# References

1. http://www.loc.gov/standards/iso639-2. A URL reference to ISO-639-2/T language codes.
2. http://www.w3.org/TR/NOTE-datetime. A URL reference to the ISO 8601 time format.
3. http://www.id3.org/. A URL reference to the ID3 metadata format.
4. http://android.git.kernel.org/?p=platform/external/opencore.git;a=tree;f=doc;hb=master "OMX Core Integration Guide".

# Table of Contents

# List of Figures

# 1 Introduction

This document provides detailed information for developers writing clients to the PVPlayer SDK. Information covered includes an overview of the high-level architecture, a description of control flow and data flow, details of the state machine, error handling, asynchronous events, and use-case scenarios. The document also covers the topic of logging and diagnostics.

## 1.1 PVPlayer SDK Definition

PVPlayer SDK is a set of components and modules that allows synchronized playback of multimedia presentations. A multimedia presentation is defined as a collection of various media that are rendered together in some sort of a synchronous manner. This could be in the form of a file encoded into a specific format (like MP4, 3GPP), a live RTSP streaming session, or a SMIL presentation or any other form.

In addition to standard playback features such as repositioning and volume control, PVPlayer SDK offers more sophisticated features such as downloading of content, and playback of content as it is being downloaded. The amount of features contained in a particular PVPlayer SDK depends on the requirements, design decisions, and limitations imposed by the platform and the chosen design.

## 1.2 PVPlayer SDK Scope

PVPlayer SDK includes all components needed to satisfy the definition above, but excludes the application (graphical or command-line) which uses the PVPlayer SDK, the operating system or platform that PVPlayer SDK runs on, the data sources (e.g. multimedia file, streaming server), and the sinks (e.g. audio device, display) for the multimedia presentation. The scope of PVPlayer SDK could be further reduced for particular platform with particular feature sets, but this document covers the largest extent of PVPlayer SDK. PVPlayer SDK is composed of and utilizes other components from PacketVideo (e.g. OSCL, PVMF nodes) so certain details might be referred to another document.

## 1.3 Audience

This document is intended for people wanting to understand what is PVPlayer SDK and for developers working on or using PVPlayer SDK. Information contained within this document will allow people to know what PVPlayer SDK can and cannot do, to learn how to use PVPlayer SDK, and to modify PVPlayer SDK for new features or debug problems.

# 2 High Level Design

## 2.1 Scope and Limitations

The PVPlayer SDK incorporates all the necessary features to support the requirements listed in the previous section. The set of features is designed to handle the requirements of a fairly complete player application. The modular architecture and designed extension mechanism provide convenient mechanism for expanding or customizing the feature set when necessary. Even between new releases and upgrades of the PVPlayer SDK, it is possible to customize certain behavior through the components that are passed to the PVPlayer SDK from the outside (e.g., the sources and sinks).

## 2.2 Requirements on Platform and Tools

The design and implementation of the PVPlayer SDK imposes certain requirements on the platform/operating system and the development tools. The PVPlayer SDK is written in the C++ language so it requires ANSI C++ development tool support for the platform. The player implementation does not require every feature defined by the C++ standard. For example, run time type indication (RTTI) is not required nor is exception handling. However, C++ template support is required. If the PVPlayer SDK interface is expected to provide another type of interface (e.g. C, Java), PVPlayer SDK can provide an adaptation layer interface but the internal components still need to be compiled in C++.

The PVPlayer SDK source code is based on PacketVideo's Operating System Compatibility Library (OSCL), the PacketVideo Multimedia Framework (PVMF) and the OpenMax Integration Layer (OMX IL 1.x) components. The PVPlayer SDK relies on OSCL to provide system functionality that is portable across platforms (i.e., it serves as an OS abstraction layer that presents a platform-independent API to the PVPlayer SDK). PVMF is the framework defining the multimedia architecture upon which the PVPlayer SDK is based. OSCL requires a platform with services provided by fairly complete operating system. The platform must have services such as dynamic memory management, threading, file I/O, network sockets, domain name services, and time information. For a complete list of platform services expected by OSCL, refer to the OSCL design and porting documents.

All PV codecs are wrapped with the OMX IL interface which is an open standard defined by the Khronos group (www.khronos.org). PVMF only communicates with codecs through the OMX IL APIs. This interface facilitates integration with 3rd party codecs as well as the PacketVideo SW codecs resident in the SDK. It is assumed that users of the PVPlayer SDK are familiar with the principles defined and referenced in the "OMX CORE Integration Guide"

## 2.3 Architecture and Component Breakdown

The PVPlayer SDK architecture follows the standard architecture defined by PVMF with a modular structure that makes the SDK flexible, scalable, and portable. The PVPlayer engine is the heart of the PVPlayer SDK. The engine utilizes PVMF nodes and node graphs to process data and internal utilities for node registration, discovery, and graph construction. The interface to the PVPlayer engine can be the primary OSCL-based one or it can be adapted to another specification based on the platform requirements. The diagram below shows a typical composition of the PVPlayer SDK. The actual composition would differ from one platform to the next so optional components are colored in yellow. If the adaptation layer were not present, the application would interface directly with PVPlayer engine and PVMF nodes.

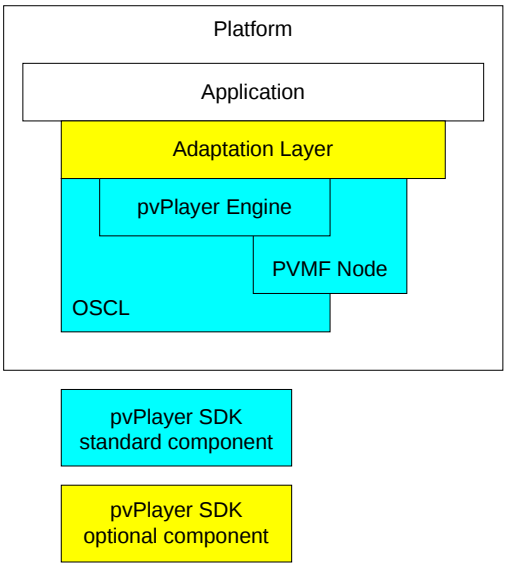



Figure 1: PVPlayer SDK Software Stack

## 2.4 Control Flow

Playback control for PVPlayer SDK originates from the user of the PVPlayer, typically a player application. The player application is responsible for instantiating and destroying PVPlayer SDK and calling the appropriate PVPlayer SDK APIs to initiate, handle, and terminate multimedia playback.
Within PVPlayer SDK, control flow is usually top-down. The application requests are received by PVPlayer engine via adaptation layer if present. The PVPlayer engine then sends the appropriate control data to PVMF nodes that it utilizes. There are some control data between connected nodes but major control data is between PVPlayer engine and PVMF nodes.

## 2.5 Data Flow

The PVPlayer SDK processes multimedia data by using one or more PVMF nodes connected together in a graph. The types of PVMF nodes used and the graph configuration would depend on the playback parameters such as source clip type and playback operation. Other types of data such as clip metadata and performance profile would be extracted by PVPlayer engine or PVMF node or combination of both and then returned to the user of PVPlayer SDK through the appropriate interface.

# 3 PVPlayer Engine Design

The PVPlayer engine is the heart of PVPlayer SDK. It receives and processes all requests for PVPlayer SDK from the user and manages the PVMF components required for multimedia playback and related operations. The idea is to hide the details of direct interaction with the multimedia components from the application and simplify its task to high-level control and status. The PVPlayer engine also detects, handles, and filters events and information generated during multimedia playback operations.

## 3.1 PVPlayerInterface API

Users of all PVPlayer SDK interfaces to PVPlayer engine via an interface class called PVPlayerInterface regardless of whether there is an adaptation layer interface between the user and PVPlayer engine. PVPlayerInterface is an OSCL-based interface and follows the common interface design for PacketVideo SDK. In addition to multimedia playback specific APIs, PVPlayerInterface provides methods to retrieve SDK information, manipulate logging, and cancel commands. To expose other interfaces available from PVPlayer engine based on PVPlayer SDK configuration and current runtime status, PVPlayerInterface provides methods to query and retrieve extension interfaces. For a list and description of PVPlayerInterface API, refer to the PVPlayerInterface API document generated from doxygen markup.

## 3.2 Asynchronous Operations

The PVPlayer engine processes most commands initiated by API calls asynchronously. There are some commands that are processed synchronously and they can be differentiated by the return value. Synchronous commands return a PVMF status code which tells the user whether the command succeeded or not and if it did fail, what the error was. All asynchronous commands return a command ID. For the user to be notified of asynchronous command completion, the user must specify a callback handler when instantiating PVPlayer engine via the factory function. When the asynchronous command completes, PVPlayer engine calls the callback handler with the command ID for the command, command status, and any other relevant data. To process the command asynchronously, the PVPlayer engine is implemented as an active object, which gets to run according to the active scheduler running in the thread. The PVPlayer engine expects scheduler to be available when instantiated and the engine itself will not directly create a thread or scheduler.

With asynchronous commands, there is a possibility of commands not completing in expected time. To deal with this issue, PVPlayer engine provides standard PV SDK APIs to cancel a specific or all issued commands. The user of PVPlayer SDK can use these APIs to cancel any request that did not complete in time or are not needed due to changing circumstances. In PVPlayer engine, it might have to deal with lower level components that behave asynchronously. To prevent an unresponsive lower level component from blocking PVPlayer engine operation, PVPlayer engine has timeout handling for any asynchronous commands that it issues. When timeout does occur, the asynchronous command is canceled and is handled appropriately (e.g. command failure, error event).

## 3.3 Event Handling

The PVPlayer engine notifies the user of errors and other information not related to API calls as unsolicited events. The notification is handled by making a callback on handlers specified by the user of PVPlayer engine. There are two callback handlers, one for error events and one for informational events, that must be specified by the user when instantiating PVPlayer engine via the factory function.

## 3.4 PVPlayer Engine Structure

The component diagram below illustrates how the PVPlayer engine interfaces to the application when the application uses PVPlayerInterface directly without any adaptation layer. PVPlayerFactory component handles the instantiation and destruction of PVPlayerEngine object. All PVPlayer engine APIs are provided by PVPlayerInterface. PVPlayerEngine uses the three callback handlers passed in by the application, PVCommandStatusObserver, PVInformationalObserver, and PVErrorEventObserver, to notify the application above asynchronous command completion and unsolicited error and informational events.

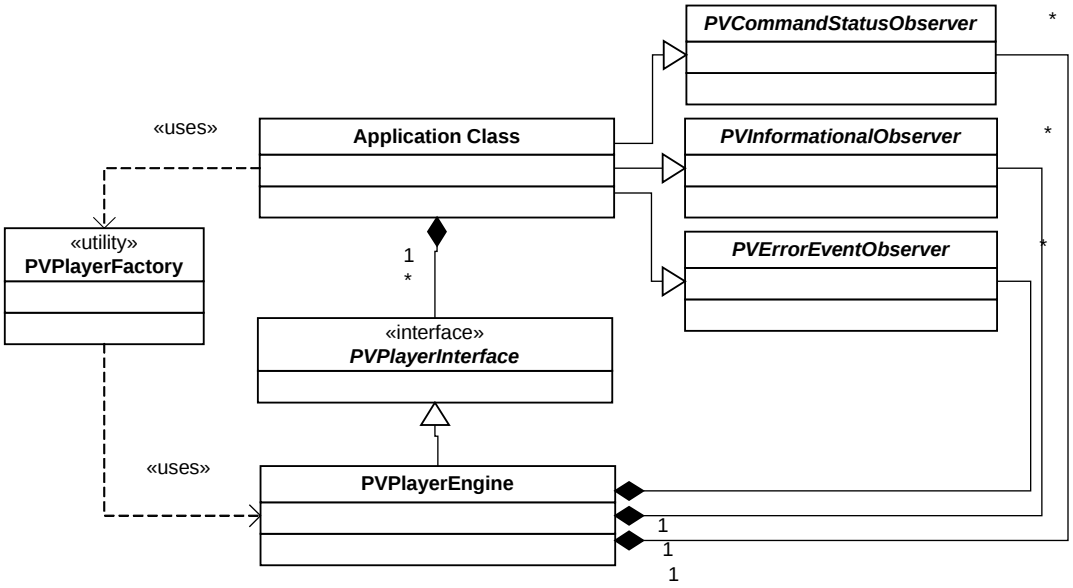



Figure 2: Class Diagram

## 3.5 State Transition Diagram

PVPlayer engine maintains a state machine and the state is modified based on PVPlayerInterface APIs called and events from PVMF components below. The diagram below shows the state transition diagram for PVPlayer engine's state machine.

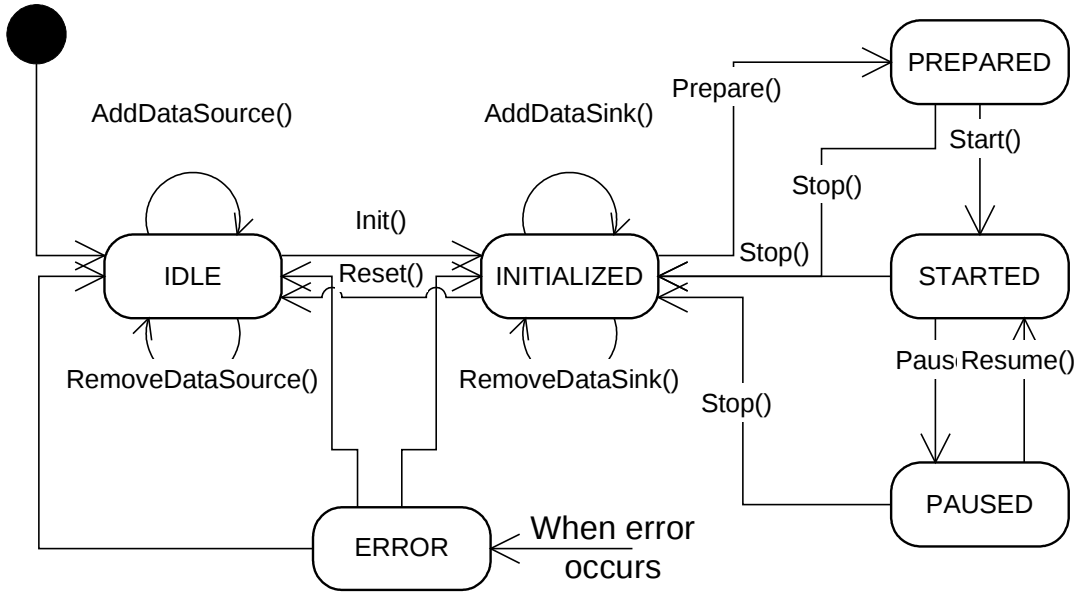



Figure 3: State Transition Diagram

The PVPlayer engine starts in the IDLE state after it is instantiated. While in the IDLE state, the data source(s) for multimedia playback can be specified by AddDataSource() API. After the data source is specified, calling Init() puts PVPlayer engine in INITIALIZED state which means the data source has been initialized. In the INITIALIZED state, the user can query data source information such as available media tracks and metadata. While in INITIALZED state, the user calls AddDataSink() to specify the data sink(s) for multimedia playback.

After all the data sinks are added, the user calling Prepare() causes PVPlayer engine to set up the necessary PVMF nodes in a data-flow graph (the data-flow graph is covered in later section) for multimedia playback based on data sources and data sinks specified. Media data is also queued for immediate playback in PREPARED state. The user calling Start() in PREPARED state initiates the actual multimedia playback and PVPlayer engine goes to STARTED state. Media data flows from data source to data sink and out of the sink in a manner specified by the user. The user can go back to the INITIALIZED state from the PREPARED state by calling Stop(). Doing so would have the PVPlayer engine stop the data-flow graph and flush all queued media data.

While the engine is in STARTED state, the user can either call Pause() or Stop(). Calling Stop() immediately ceases playback operation, flushes all media data, and places the engine back in INITIALIZED state. If Pause() is called, playback operation is stopped but media data in the flow is not flushed. PvPlayer engine goes into PAUSED and playback operation can continue from where it paused by calling Resume(). Stop() can also be called from PAUSED state to return the engine to the INITIALIZED state.

Calling Stop() returns PVPlayer engine to the INITIALIZED state. Back in the INITIALIZED state, data sinks can be added and/or removed by calling AddDataSink() and RemoveDataSink(). Playback can restarted by calling Prepare() then Start(), but to go back to the IDLE state for shutdown or to open another data source for playback, the user must call Reset(). If all data sinks are not removed by explicitly calling RemoveDataSink() in INITIALIZED state, Reset() call removes all the data sinks. After Reset()

completes, the engine is back in IDLE state. Data sources can be removed with RemoveDataSource() and new data sources can be added with AddDataSource(). If the user wants to shutdown PVPlayer SDK, PVPlayer engine can be properly destroyed in the IDLE state. It is also possible to call Reset() while in PREPARED, STARTED, or PAUSED state. Internally this will trigger a Stop() call followed by a Reset().

If PVPlayer engine encounters an error due to usage error or error events from within or components below which requires time to properly handle, the engine will go into a transitional ERROR state and try to recover. If the error is unrecoverable or if the engine encounters more errors during error recovery, PVPlayer engine will clean up everything and go to the IDLE state. If the engine recovers from the error, the resulting engine state would depend from which state the engine encountered the error. If the engine was in or past the INITIALIZED state (PREPARED, STARTED, PAUSED, or any transition state in between), PVPlayer engine will try to recover to the INITIALIZED state. If the error occurred while in IDLE or initializing, then PVPlayer engine will try to recover to the IDLE state without performing a total cleanup. When error recovery completes, PVPlayer engine will report PVMFInfoErrorHandlingComplete informational event. To determine whether the engine is handling the error asynchronously, the user should check the state of the engine synchronously in the command completion or error event handler. If the engine state is the ERROR state, the user should wait for the PVMFInfoErrorHandlingComplete informational event.

This state transition diagram describes the basic state transition model for all PVPlayer engine playback operation.

# 4 Interface

## 4.1 Default Interface

The standard interface to PVPlayer engine interface is the OSCL-based interface, PVPlayerInterface. This is the base level API which directly controls PVPlayer engine. Use of this interface requires the user to be aware of OSCL types and components and PVMF types and components.

## 4.2 Adaptation Layer

If the interface to PVPlayer SDK needs to be different than the OSCL-based interface, another interface layer needs to be created to “wrap” around the OSCL-based interface. This “wrapper” is referred to as an adaptation layer for OSCL-based PVPlayer engine interface.

One possible reason to create an adaptation layer would be to encapsulate the OSCL interface with types and components of a particular platform or operating system (e.g. ANSI C interface, Symbian interface). Another reason would be that the adaptation layer modifies the interface and behavior of PVPlayer SDK to match the expectation of the application (e.g. legacy interface). The adaptation layer could also combine PVPlayer SDK with another SDK or component to provide a unified interface to the application. The block diagrams below illustrate how the adaptation layer relates to PVPlayer Engine and its OSCL-based interface. The diagram on the right shows the adaptation layer adding more functionality by including another engine.

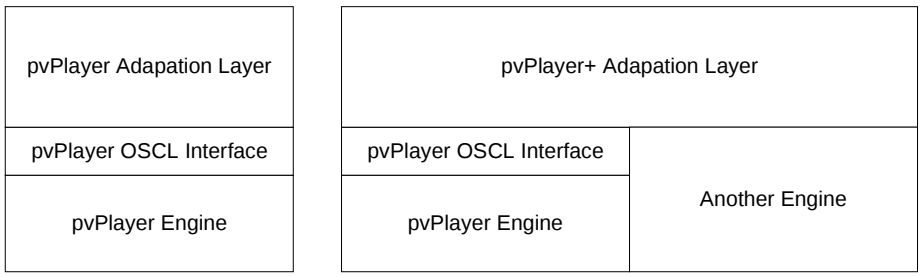



Figure 4: PVPlayer Adaptation Layer

## 4.3 Multi-Threading Support

The default OSCL-based interface can also be used across multiple threads safely with some precautions. To have multi-threading support in the interface, the application needs to launch a thread within which it would initialize OSCL, create an OSCL-based PVPlayer engine instance, and then start the scheduler. The application, at this point, can use the PVPlayer engine instance pointer and make API calls from any thread. Note that the Synchronous calls will block till completion. The asynchronous command completion callsbacks, and the Informational/Error events callbacks will happen from the engine thread.

## 4.4 Media Data Output to Data Sink

The PVPlayer engine can utilize any PVMF node as the media data sink, but in most PVPlayer SDK usage, synchronized media data would be rendered via appropriate output media devices. For video, the media device would be the display and for audio, the media device would be the PCM audio device. Output media devices are typically platform specific. PvPlayer SDK handles interfacing to platform specific output media devices one of two ways. First method is to encapsulate the media device in a PVMF node which PVPlayer Engine can use directly. This method minimizes the code between PVPlayer Engine and the media device interface, but requires a new PVMF node to be created. The second method is to interface the media device to PV's Media I/O interface. By encapsulating the media device in PV Media IO interface, PVPlayer Engine can use the PVMF node that interfaces PV Media IO to output the media data. PV's Media I/O interface is less complex than PVMF node and specific for media output, but this method adds layers and code. The diagram below shows the two methods in relation to PVPlayer Engine. For more information on PV Media IO interface, please refer to the PV Media IO documents.

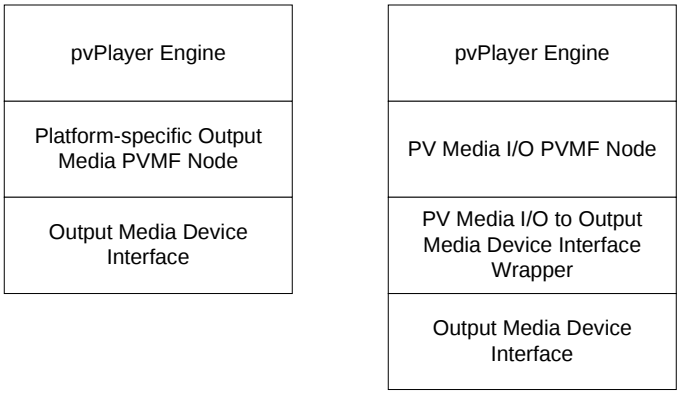



Figure 5: Media Output to Node and Media IO

## 4.5 Porting to a New Platform

Porting for PVPlayer SDK is having PVPlayer SDK working on a particular platform. Since PVPlayer engine is strictly OSCL-based, porting for the engine would be accomplished by adding support for particular platform in OSCL.
Porting rest of PVPlayer SDK would depend on the configuration of the SDK. If the configuration is all OSCL-based including nodes and data sources/sinks, porting would be accomplished by porting OSCL. If the configuration requires usage of platform specific components like hardware accelerators and particular decoder interfaces, a new node would need to be created to encapsulate the use and register the new node for the PVPlayer engine to use. If the data source and/or sink are platform specific, new PVPlayer data source/sinks needs to be created to encapsulate the platform dependency and the user of the PVPlayer engine (adaptation layer or application) would need to pass it in.

# 5 Temporal Synchronization

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 do so, PVPlayer SDK relies on information from a playback clock, timestamps from the media data, and optionally timing information from data sinks that accept media data in a specified rate (e.g. audio device set at fixed sampling rate). PvPlayer SDK’s temporal synchronization also allows the playback speed to be adjusted and this feature would also be described in the following sections.

## 5.1 Clock in PVPlayer SDK

The PVPlayer SDK uses a clock in PVPlayer engine to determine the temporal playback rate. The playback clock is based on PVMF media clock which provides a control to set, start, pause, stop, and adjust the clock. PVMF media clock also allows the timebase to use for the clock source to be specified by PVPlayer engine. For more information PVMF media clock, please refer to its design document.
PvPlayer engine creates an instance of PVMF media clock to keep track of the playback clock. PvPlayer engine is responsible for changing the state of the clock due to changes in playback operation (start, pause, resume, stop).

The playback clock used in PVPlayer engine is non-decreasing during playback. This means the playback clock never goes back even if the playback repositions to an earlier time. The playback clock does not represent the actual position in the clip which is called normal playback time or NPT. To return NPT to the user of PVPlayer SDK, PVPlayer engine always maintains a mapping between NPT and playback clock time.

A reference to this clock is passed to data sinks which require a clock to perform synchronization of media data. Description of how the data sinks use the clock for synchronization is presented next.

# 6 Synchronization with timestamps

For data sinks with passive rendering, PVPlayer engine must output the media data properly in time. This is accomplished by evaluating the timestamp associated with each media data with the current value of the playback clock. If the media data's timestamp is equal to the current clock time, the media data is in synchronization and rendered. If the timestamp is less than the clock time, the media data is early. If the media data is greater than the current clock time, the media data is late. What happens to media data that is early or late depends on how PVPlayer engine is configured. In most uses, early media data is held until it becomes in synchronization and late media data is dropped without being rendered. But in some configurations, the late media data might be rendered as well.

Another area in timestamp synchronization that could be configured is the margin for being in synchronization. In ideal situation, the margin is 0 where timestamp must be equal to the clock time to be in synchronization. But due to various factors such as clock resolution and active object scheduling resolution, the margin must be larger or the media data being in synchronization would be missed. The synchronization point could also be offset to deal with some fixed latency in the rendering. For example, if a video render device requires certain time to actually display the video frame, the synchronization might happen at an earlier time so when data is sent to the device and the actual display would occur when the timestamp value equaled the clock time.

This synchronization functionality is performed in the data sinks with such support. These data sinks take a reference to the playback clock from PVPlayer engine and reads the media data timestamp from each PVMF media data object. PvPlayer engine determines if a data sink has synchronization support in the capability exchange process.

## 6.1 Synchronization with flow controlling data sink

If the data sink has flow control and the media data for that sink is rendered continuously, PVPlayer engine needs to take the data output rate of the data sink into account. If there is a temporal difference between how the flow controlled media data is actually rendered by the data sink and PVPlayer engine's playback clock, AV synchronization mismatch might appear between the media tracks. Depending on the severity of the mismatch, the problem might be detectable by the person viewing the multimedia playback.

To prevent such a temporal difference, the flow controlled data sink would perform adjustments to the playback clock based on information of media data rendering. Such information could be fed back from the rendering device on how much of data has been actually rendered or the time of last rendered media data. The data sink is responsible for converting the correction information to a format that acceptable to the clock adjustment method of PVMF media clock. An example of a data sink with flow controlling rendering is a PCM audio output sink node. PCM audio data is continuous and audio devices typically output the PCM audio data by some fixed sampling rate. The clock controlling the output could be different from the clock source that PVPlayer engine's playback clock uses so there could a difference in how time progresses between the two. Over time, the difference could accumulate and other media data (e.g. video, text) could be rendered out-of-synchronization with the audio. Audio devices could return the number of PCM samples rendered or the system time when the last audio media data was rendered and the audio output data sink node could use this information to adjust PVPlayer engine's playback clock.

## 6.2 Synchronization with combination

In typical multimedia playback scenarios, PVPlayer SDK will interact with data sinks of both type: one which relies on timestamp only and one which relies on flow control. In such cases, the data sink with flow control is allowed to adjust the playback clock so all media data is kept in synchronization with each other.

## 6.3 Faster or slower than “real-time”

Since all media output rate in PVPlayer SDK is controlled by PVPlayer engine’s playback clock, the playback rate can be changed by modifying the pacing of the playback clock. The playback clock is based on PVMF media clock class so it uses a timebase to know how much time has elapsed. Typically the timebase uses the system tickcount or some other system timing function to report how much time has elapsed in microseconds. By using such a timebase, PVPlayer SDK will playback the media data in “real-time”. But if the timebase was modified to report elapsed time as being faster than or slower than “real-time” then playback could occur faster or slower respectively. By making such modifications to the timebase, PVPlayer SDK provides such features as fast forward (faster than “real-time”), slow motion (slower than “real-time”), or frame-by-frame (slower than “real-time” without set rate).

When playback rate is modified as such, media tracks with data sinks that can only work in one fixed rate must be disabled since those sinks cannot play the data faster or slower. Typical data sink with such a limitation is the audio output device data sink. The audio output device usually can only accept audio data in a fixed sampling rate. If the playback rate changed, the audio data would be fed to the device too fast or too slow and could cause the data sink to overflow or underflow with undesirable effects. Therefore in such a case, PVPlayer engine will disable media tracks with data sinks with such restrictions. PvPlayer engine will determine if the data sink can handle different playback rates by querying its capabilities.

# 7 Playback Control

PVPlayer SDK provides methods to control the multimedia playback. This section describes what occurs inside PVPlayer engine when these control commands are issued.

## 7.1 Starting and Stopping

When starting playback, PVPlayer engine commands the PVMF nodes in the data-flow graph to start and then starts the playback clock. When stopping, PVPlayer engine stops the playback clock and commands the PVMF nodes in the data-flow graph to stop. Doing so flushes all media data in the data-flow graph.

## 7.2 Pausing and resuming

When paused, the playback clock is not progressing forward and media data is not rendered via the data sinks. But unlike being stopped, media data is still queued in the data-flow graph ready to be restarted. Resuming takes PVPlayer SDK out of paused state to have playback clock moving forward and media data to be rendered again.

When pausing, PVPlayer engine pauses the playback clock and commands the PVMF nodes in the data-flow graph to pause.
When resuming, PVPlayer engine restarts the PVMF nodes in the data-flow graph and then restarts the playback clock.

## 7.3 Repositioning

Repositioning is the changing of playback position in the clip during playback. An example would be to be playing the clip at 10 seconds and then immediately jumping to the clip at 30 seconds and continuing playback. pvPlayer SDK handles repositioning as a change in the data source's media data position and continuing playback. Since the playback clock does not jump during playback, the data source or PVMF node responsible for providing the timestamp for the media stamp adjusts the media data timestamp to maintain this requirement.
For example, playback has been started and currently the playback position is at 30 seconds. If the playback is repositioned to the clip's time (normal playback time, NPT) of 15 seconds, the playback clock is still kept at 30 seconds and media data will be sent from clip at 15 seconds but the timestamp will continue to be from 30 seconds. After 30 more seconds, the NPT is at 45 seconds, but the playback clock and media data timestamp would be at 60 seconds. At this point, if a forward repositioning to clip's time of 90 second occurs, media data will be sent from clip at 90 seconds but the playback clock and media data timestamp will still be at 60 seconds. The table below lists this sequence.

| Event | Playback clock | Clip time (NPT) |
|---|---|---|
| Start playback | 0 | 0 |
| Playback for 30 sec | 30 | 30 |
| Reposition to 15 sec in clip | 30 | 15 |
| Playback for 30 sec | 60 | 45 |
| Reposition to 90 sec in clip | 60 | 90 |
| Playback for 30 sec | 90 | 120 |

The example above represents an ideal repositioning scenario. When repositioning with some data sources, PVPlayer SDK would not be able to directly reposition to specified position due to media data limitations or data source restrictions. Example of such limitation is the time resolution of the media data (e.g. audio frame) and limited seek positions (e.g. I-frames in video). If PVPlayer SDK is handling such data, there could be a transition period in reposition where additional media data might be generated and processed. PVPlayer SDK would behave to minimize such transition period but some artifacts of the transition might be unavoidable.

The PVPlayer SDK could handle the reposition transition in one of several ways and the SetPlaybackRange() API provides options to configure this. One such configuration deals with repositioning for video with limited seek positions (e.g. M4v). In such video data, one cannot go and playback from any frame since frames are dependent on the previously decoded frame. Playback has to start from certain frames which are not dependent on previous frames. So when repositioning, if jump-to location is at one of these frames that are not dependent on the previous frame, then playback can continue from that frame. If not, then PVPlayer SDK must go to one of these non-dependent frames before the requested repositioning position and decode the frames in between before continuing playback. For best quality, PVPlayer SDK should always go to one of these independent video frames. But if availability of these independent frames are limited, PVPlayer SDK might take some time to decode the in-between frames. In such case, the better user experience might be to just decode from a dependent frame at the requested repositioning point while sacrificing video quality. To allow the PVPlayer SDK behavior for the transition to be configurable by the user, the SetPlaybackRange() API provides a way to configure whether to always go to the independent frame or not via one of the optional parameters of the API, namely, "*aSeekToSyncPoint*". The default value of this boolean is ***true***, which means that the PVPlayer SDK would always start from one of the independent video frames. To disallow this, and to start from the requested repositioning point, the user can set this boolean to ***false.*** The size of the window to look for the independent frames can be set via the capability-and-configuration interface (refer to the next section). If not always going to the independent frame, playback will start from a dependent frame unless there is an independent frame at the requested repositioning position. If always going to independent frame and the window is non-existent, then PVPlayer SDK will always look for the independent frame that is before the requested repositioning position. Between those two extremes, if the independent frame falls in the specified window then repositioned playback will start there. If such a frame is not found in the window, the first dependent frame past edge of the window would be used as the starting point for the repositioning. The diagrams below illustrate how the windowing works. In the first diagram, the independent frame (sync point) is outside of the window so PVPlayer engine will reposition to the edge of the window (new position).

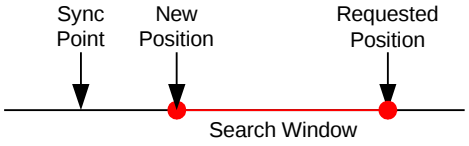



Figure 6: Independent Frame is Outside of Window

In the second diagram, the independent frame (sync point) is inside of the window so PVPlayer engine will reposition to the same position (new position).

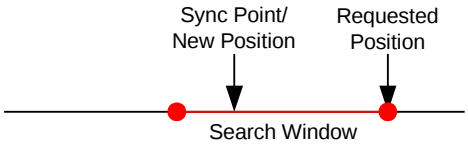



Figure 7: Independent Frame is Inside Window

The flow chart below describes how repositioning would be performed by PVPlayer engine based on the reposition configuration.

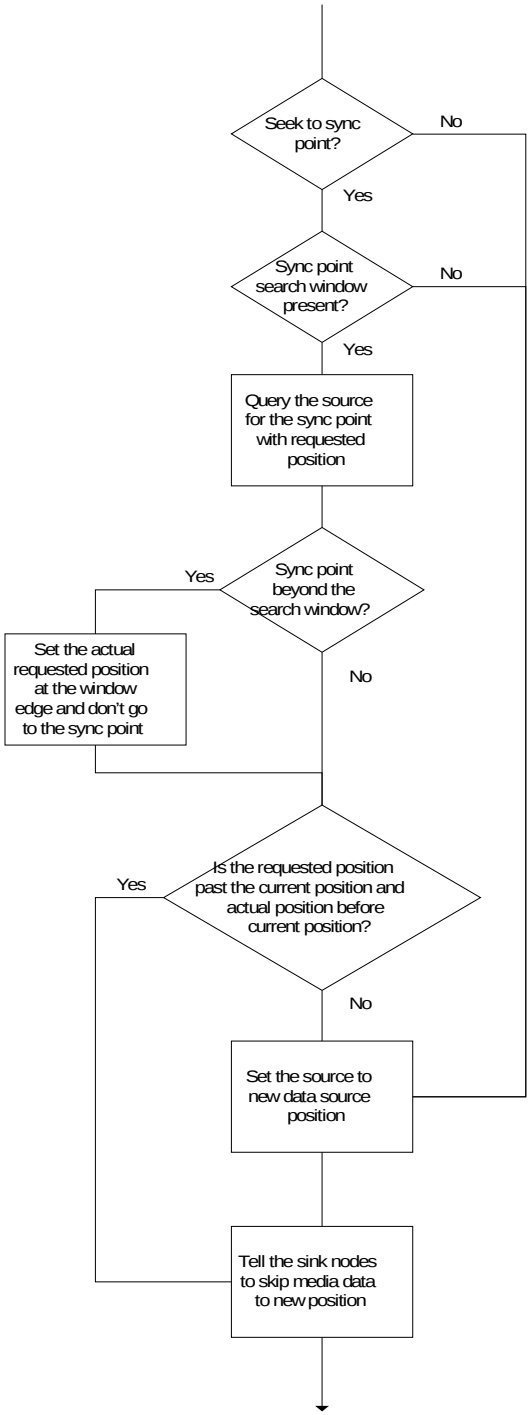



Figure 8: Reposition Processing Flow Chart

The reposition flow chart also incorporates a forward repositioning optimization when always going to an independent frame and the search window exists. If the requested repositioning position is past the

current playback position and the position of the independent frame is before the current playback position, PVPlayer engine will not modify the source data position. This optimization saves unnecessary data transfer and processing between the independent frame position and the current playback position when repositioning.

Another configurable repositioning behavior which is not show in the flowchart is the option to always start rendering the media data at the requested reposition position. By default, the PVPlayer engine will always tell the sink nodes to skip media data up to the requested reposition position irrespective of where the source node starts sending the media data from. This is governed by the boolean parameter "*aSkipToRequestedPosition*" of the SetPlaybackRange() API, which is set to ***true*** by default. If this is turned off, then the sink node will try to render all media data that it receives as long as it is received on time.

As mentioned above, the default values of *aSkipToRequestedPosition*(true) and *aSeekToSyncPoint*(true) is what is ideal for a normal playback application. An example where these booleans can be tweaked for an application is a "Video Scrubber". For a "Video Scrubber", the basic requirement is to display a legible frame as fast as possible, and at a point close to the requested position. So, here the application could retain *aSeekToSyncPoint* to ***true*** and change *aSkipToRequestedPosition* to ***false***. a*seekToSyncPoint* is set to true because we want a independently decodable frame, and *aSkipToRequestedPosition* is set to false because we don't want to lose time by skipping frames.

Another deviation from the ideal repositioning scenario is the repositioning occurring when there are media data in the data-flow graph that are waiting to be rendered by the data sinks. This occurs if the nodes in the data-flow graph process media data ahead of the playback rate to have the media data for rendering in time and the repositioning command is not known beforehand. When repositioning, these media datas become obsolete and should not be rendered. To prevent these obsolete media data from being rendered, PVPlayer engine queries the data source node for the starting timestamp for media data after repositioning. The data source node knows this information since the data source node controls the media data going into the data-flow graph. The data source node calculates the repositioning timestamp as the next time value past the last media data sent into the data-flow graph. Then the PVPlayer engine stops and sets the playback clock to this starting timestamp and commands all data sink nodes with synchronization support to flush media data before the new repositioning timestamp. The playback clock is started again when all data sink nodes report old media data has been flushed. With this feature, the repositioning example for the ideal case would be changed to the following:

| Event | Playback clock | Clip time (NPT) |
|---|---|---|
| Start playback | 0 | 0 |
| Playback for 30 sec | 30 | 30 |
| Reposition to 15 sec in clip but 2 sec worth of data in data-flow graph | 32 | 15 |
| Playback for 30 sec | 62 | 45 |
| Reposition to 90 sec in clip but 4 sec worth of data in data-flow graph | 66 | 90 |
| Playback for 30 sec | 96 | 120 |

Repositioning use cases can be divided into three categories. First one is offset playback where the starting position is known before playback starts. Second one is "edit-list" playback in which when and where to reposition are known beforehand. The last use case is random positioning playback in which where to reposition is known but when to reposition is not known until the command is requested.

In the OSCL-based interface to PVPlayer engine, all three repositioning types can be realized by the SetPlaybackRange() API. Depending on the PVPlayer engine state when SetPlaybackRange() command is issued and the parameters passed in, the PVPlayer SDK user can perform all three repositioning types in playback.
SetPlaybackRange() API has two parameters for the beginning playback position and ending playback position. When the command is accepted, PVPlayer engine will play the media data between these two positions. SetPlaybackRange() also has a flag to specify whether to activate the new range immediately or queue for activation later when the current playback range completes. PvPlayer engine can only queue one playback range at any given time so if multiple playback ranges are queued at once, only the last one queued will be actually activated. For offset playback and random positioning use cases, the flag is set to immediate activation. For “edit-list” playback, the flag is set for queuing.

Begin and end playback position parameters are allowed to be indeterminant. In such case, the beginning of the clip (time 0) and end of clip (clip duration) will replace begin and end positions, respectively. The only exception is when SetPlaybackRange() is called during playback with begin position being indeterminate and flag set for immediate activation. In such scenario, the end position will be modified without interrupting the playback (i.e. will not random position to beginning of clip).

SetPlaybackRange() can be called in most engine states, however, for performance reasons it should be called as early as possible. For example if playback is to start from 30 seconds instead of 0 seconds it would be possible to reposition after Init, after Prepare or after Start has been issued. The later two cases are inefficient because data has already been retrieved and processed and now also needs to be flushed. In this case it would be best to call SetPlaybackRange() before calling Prepare() to avoid unnecessary calculations and improve start up speed. When SetPlaybackRange() is called in the paused state, the pvPlayer SDK releases one frame of video to the video MIO. This frame corresponds to the new position in the clip from which playback will resume.

# 8 Capability Query and Configuring Settings

PVPlayer engine utilizes the PVMF capability-and-configuration interface to allow the application to access and modify engine and node settings not exposed by the player interface. The extension interface (PvmiCapabilityAndConfig) is exposed via the player API, QueryInterface(), by requesting with the UUID associated with the interface. Using the returned interface pointer, the application can query, verify, and set settings at the engine and node levels. At the node level, the node being used by the engine must support the capability-and-configuration interface as well for node settings to be accessible to the application.

Capability-and-configuration interface uses key strings in PacketVideo Extended MIME String (PvXms) format to specify the settings of interest. PvXms extends the standard MIME string format by allowing additional levels of subtype strings all separated by the slash character. Using key strings adds complexity in parsing but allows flexibility and extensibility for settings without greatly modifying code when settings are added, removed, or modified. In addition to specifying the setting of interest, the key string also provides information on value returned with the string in a key-value pair (KVP). The “type” parameter in the key string tells the user of the KVP whether there is a valid value if “type=value”. The “valtype” parameter in the key string tells the user of the KVP what the value type is so the appropriate union member can be accessed in the KVP.

## 8.1 PVPlayer Engine Key Strings

All key strings at the PVPlayer engine level start with “x-pvmf/player”. The following key strings are currently supported in the player engine:

| Key Strings With Value Type | Description |
|---|---|
| x-pvmf/player/pbpos_units;valtype=char* | Playback position units specified with strings (“PVPPBPOSUNIT_MILLISEC”, “PVPPBPOSUNIT_SEC”, “PVPPBPOSUNIT_MIN”, “PVPPBPOSUNIT_FILEOFFSET”) |
| x-pvmf/player/pbpos_interval;valtype=uint32 | The interval between playback position info event. Integer value in milliseconds. |
| x-pvmf/player/renderskipped;valtype=bool | The flag to specify whether to render the skipped frames |
| x-pvmf/player/silenceinsertion_enable;valtype=bool | The flag to specify whether to check for any gaps in an audio bitstream and insert silence samples. |
| x-pvmf/player/syncpointseekwindow;valtype=uint32 | If seeking to closest sync point, this parameter specifies how far to search back in milliseconds. If the sync point is not present in the specified window, playback would continue from the window boundary. Value of 0 means no window. |

## 8.2 Node Level Key Strings

The node level key strings available during PVPlayer engine usage depends on PVMF nodes being used by the PVPlayer engine at that time and the key strings supported by a particular node. For node level key strings, PVPlayer engine acts as a router to pass any requests to the appropriate node. Currently, PVPlayer engine performs a hardcoded mapping from key sub-string to certain nodes, but in the future, PVPlayer engine and nodes will determine the mapping at runtime using a registration scheme.

Currently, the key string mapping to nodes is as follows in PVPlayer engine.

| Key Sub-String | Node Type |
|---|---|
| x-pvmf/video/decoder | Video decoder node then video sink node |
| x-pvmf/audio/decoder | Audio decoder node than audio sink node |
| x-pvmf/video/render | Video sink node |
| x-pvmf/audio/render | Audio sink node |
| x-pvmf/net | Data source node (typically streaming / download) |
| x-pvmf/parser | Data source node (typically local playback sources) |

PVMF video decoder node key strings are listed below. The key strings allow settings associated with M4v and H.263 video decoding to be queried and modified when PVMF Video Decoder node is used to decode video bitstreams to YUV.

| Key Strings With Value Type | Description |
|---|---|
| x-pvmf/video/decoder/postproc_enable;valtype=bool | Flag to enable/disable postprocessing in video decoder |
| x-pvmf/video/decoder/postproc_type;valtype=bitarray32 | If postprocessing is enabled, the postprocessing types enabled. |
| x-pvmf/video/decoder/key_frame_only_mode;valtype=bool | Flag to enable/disable key frame only mode, in which non-key frames are skipped |
| x-pvmf/video/decoder/skip_n_until_key_frame;valtype=uint32 | If greater than 0, enables skip_n_until_key_frame mode as well as sets the maximum number of frames to skip while waiting for the first key frame of a clip during playback. A value of 0 disables skip_n_until_keyframe mode. |

PVMF streaming and download source node key strings are listed below.

| Key Strings With Value Type | Description |
|---|---|
| x-pvmf/net/delay;valtype=uint32 | Specifies the jitter buffer duration in milliseconds (typically used in streaming sessions) |
| x-pvmf/net/user-agent;valtype=wchar* | Specifies the user agent string in unicode |
| x-pvmf/net/keep-alive-interval;valtype=uint32 | Specifies the keep-alive-interval in milliseconds (this is the frequency at which the player would send keep-alive |

| | |
|---|---|
| | notifications to the server) |
| x-pvmf/net/keep-alive-during-play;valtype=bool | Specifies whether keep-alive notifications need be sent during playback (typically keep-alive notifications are sent in a paused state) |
| x-pvmf/net/http-version;valtype=char* | Specifies the HTTP Protocol Version to be used during download / streaming. |
| x-pvmf/net/num-redirect-attempts;valtype=uint32<br><br>Optional params on key:<br>1) The key can contain a “mode=” parameter to indicate if this redirect attempts applies to streaming or download session or DLA. | Specifies the maximum number of times the client would process and act on a redirect notification from the server. |
| x-pvmf/net/protocol-extension-header;valtype=char*<br><br>Optional params on key:<br>2) The key can contain “purge-on-redirect”. This means that this protocol-extension-header will not be sent to the server in case of redirect. Example: “x-pvmf/net/protocol-extension-header;valtype=char*;mode=streaming;purge-on-redirect”<br>3) The key can contain a “mode=” parameter to indicate if this extension header applies to streaming or download session or DRM.<br>Format of the value string:<br>1) The extension header is provided a s key-value pair.<br>2) The value string can contain an additional “method=” argument. This is used to specify the protocol methods to which this extension header applies. For example: “key=PVPlayerCoreEngineTest;value=Test;method=GET, POST" | Specifies any extension headers that need to be sent to the server. |
| x-pvmf/net/http-timeout;valtype=uint32 | Specifies the HTTP timeout in seconds |
| x-pvmf/net/http-header-request-disabled;valtype=bool | During progressive download player uses the HTTP HEAD request upfront to ascertain the total file size. In case it is desired that this HEAD request must not be sent then this key can be used |

| | |
|---|---|
| | to disable the same. |
| x-pvmf/net/max-tcp-recv-buffer-size-download;valtype=uint32 | Specifies the max buffer size to be used while doing recvs on the TCP socket, during a progressive download session. |
| x-pvmf/net/max-tcp-recv-buffer-size-streaming;valtype=uint32 | Specifies the max buffer size to be used while doing recvs on the TCP socket, during a streaming session. |
| x-pvmf/net/rebuffering-threshold;valtype=uint32 | Specifies the re-buffering threshold in milliseconds (typically used in streaming sessions). If the jitter buffer delay drops below this threshold, then player would enter re-buffering. This value must be less than the jitter buffer duration specified via the “delay” key string listed above. |
| x-pvmf/net/disable-firewall-packets;valtype=bool | In case of UDP streaming sessions, a firewall between the client and the server could block all UDP traffic. PvPlayerSDK attempts to unblock traffic using a proprietary algorithm, by default. This key can be used to turn off this feature. |
| x-pvmf/net/jitterbuffer-inactivity-duration;valtype=uint32 | Specifies the jitter buffer inactivity duration in milliseconds (typically used in streaming sessions). If there is no incoming media for this amount of time PVPlayerSDK will end the streaming session with an inactivity timeout error |
| x-pvmf/net/max-min-udp-port;valtype=range_uint32 | Specifies both min and max UDP port numbers (typically used in RTSP streaming for RTP/RTCP packets.) |

### 8.2.1 Download Progress Usage Detail

As discussed in Section 9.2, the application can configure the type of download progress data reported by PVPlayer using the x-pvmf/net/download-progress-info capability configuration. The application would receive download progress data when PVPlayer sends the PVMFInfoBufferingStatus event. The data can be retrieved by calling GetLocalBuffer() on the PVAsyncInformationalEvent object provided to the HandleInformationalEvent() callback. The progress data can be interpreted in the following three ways:

- If no content length is received from the server, the progress data is always the total number of bytes received from the server, regardless of x-pvmf/net/download-progress-info setting.

- If content length is received from the server, the progress data is by default the percentage of time duration of the clip that has been downloaded. For example, if 6 seconds of media data has been downloaded for a 30 second clip, the progress data would be 20%.
- If the application configures x-pvmf/net/download-progress-info setting to report progress in bytes, the progress data is the number of bytes downloaded divided by the total number of bytes in the file to be downloaded. For example, if 60KB of data has been downloaded for a 300KB clip, the progress data would be 20%.

## 8.3 Usage examples

The sequence diagram below illustrates how the application can retrieve the capability-and-configuration interface from PVPlayer engine and perform queries and changing of playback settings at the engine level and node level. Since PVPlayer engine does not support a PvmiMIOSession for the capability-and-configuration interface, NULL is passed in for interface methods with a PvmiMIOSession parameter. Also context parameter is not supported so the PvmiCapContext parameter is ignored by interface methods.

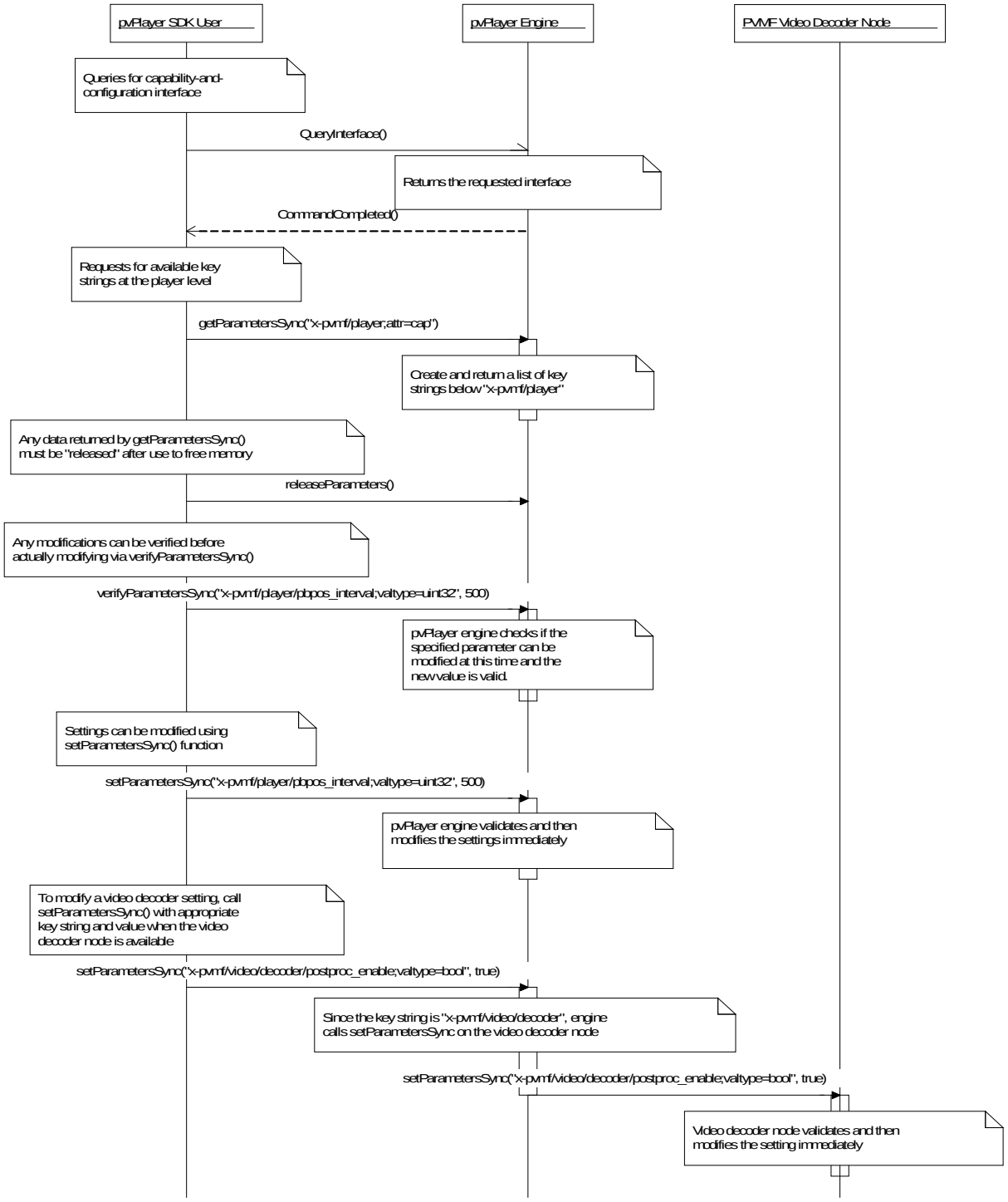



Figure 9: Capability and Configuration Interface Usage Sequence

# 9 Metadata Handling

Metadata is information about the multimedia data, which is not the media data itself. Across the different content types supported by PVPlayer, there are several different schemes defined for storing information. For example, the following is a short list of some of the metadata schemes that may be encountered in the supported file types: ID3v1, ID3v1.1, ID3v2, PV's metadata storage within an MPEG4 file, 3GPP Release 6 asset information within an MPEG4 file, and Apple iTunes metadata within an MPEG4 file. Typical information in the metadata includes such things as title, author, description, copyright, etc. The PVPlayer SDK supports retrieval of metadata by the relaying metadata queries to the underlying components that implement the actual parsing of the different metadata storage schemes.

## 9.1 Metadata retrieval APIs

Within the PVPlayerInterface, metadata is handled as key-value pairs. The API provides a way to obtain the list of available key strings and the values associated with the keys through GetMetadataValues(). Since there are usually several metadata values, the APIs use list structures for the keys and values. Also, the lists of values can be arbitrarily long, so the APIs allow segments of any size to be retrieved with each call so that it is not required to hold the entire list at once.

### 9.1.1 Metadata Related Events

In certain non-local playback metadata is not readily available with the engine. Hence it could not be retrieved at the beginning of playback. Or, in a playlist scenario, the engine does not have the metadata readily available for all the clips in the queue. In such scenarios metadata shall be fetched on the basis of informational events sent by engine.

Typical metadata related events sent by engine:

| Error Code | Meaning |
|---|---|
| PVMFInfoDurationAvailable | Duration is available, and can be retrieved now. This event itself carries the duration and there's no need to issue GetMetadataValues() api to get the duration value. |
| PVMFInfoMetadataAvailable | Metada is ready, and application can retrieve meta data now. |

## 9.2 Querying Metadata

When GetMetadataValues() is called, PVPlayer engine calls GetNodeMetadataValues() for each node that provides the metadata retrieval extension interface. As each node returns the list of requested metadata values, the metadata values are copied to the metadata value list passed in by the user of PVPlayer SDK. When all nodes return requested the metadata values, PVPlayer engine reports GetMetadataValues command as complete. Typically, the user provides a list of keys that is of interest as part of this API. If the user wishes to retrieve all available metadata keys and their values, the API allows the passing of a wild character key named "***all***".

## 9.3 Metadata Storage

Metadata is a key-value pair where the metadata key is stored as a string while the value is often a string but may be other types such as an integer, etc. The metadata value could be one of many data types so it is stored as an union of various data types. When GetMetadataValues() is called, PVPlayer engine

returns the requested values as a vector of the key-value pair structure which contains the key string along with a union containing the value. The “valtype” parameter in the returned key string specifies which of the union members to access to read the associated value. The user is responsible for parsing the key string for this “valtype” parameter to determine the data type of the value. For example a returned key string of

```
author;valtype=wchar*
```

would indicate that the returned author value is a wide-character string and is accessible in the union member corresponding to the wide character pointer.

If the metadata value is a pointer type, PVPlayer engine will allocate and deallocate the memory referenced by the pointer. But the pointer is only valid during the callback handler (i.e. CommandCompleted() function) that notifies the user GetMetadataValues command has completed. After the callback handler returns, the memory may be freed at any time and the pointer would become invalid. *Therefore, all data of interest must be copied to another storage area before the callback handler returns.*

For variable size value types (e.g. pointers such char*), the length field of the key-value pair provides the size of the valid data and the capacity field provides the total size of the buffer. Both length and capacity sizes are in the units of the type. For character string values, the length field includes the NULL terminator (e.g. if value string is "abc\0", the string is 3 characters long but length field in the key-value pair is 4). For fixed size value types, length and capacity are undefined and should not be used.

## 9.4 Metadata Keys

The different metadata schemes have variations on the exact set of information provided. Some consist of small fixed sets of information, while others like ID3v2 are extensible to allow new keys and new information in the future. However, there is a fair amount of similarity in the core set information provided by these different schemes. Therefore the PVPlayer SDK defines a set of simple common metadata keys that across the different content formats and metadata schemes. Internally, the appropriate metadata entry will be mapped to the appropriate common key. Other metadata may be accessible beyond the common set, but access to those values will use keys specific to the metadata scheme. The table below lists the set of simple common metadata keys. For example, the SDK user could query with the metadata key “author” to get the author information regardless of content type. It’s not guaranteed that a particular piece of content has any of this information stored in metadata. However the SDK user can be sure that if, for example, authoring information is stored in a supported metadata scheme, a query using the “author” will retrieve it.

Many of the keys are simple one-word strings, but the format of the key allows for more complicated forms, which may include optional parameter qualifiers. The syntax of the key strings follows a similar format to the PvXms extended MIME strings used of configuration and capability exchange within the framework, but there are some differences. For example, the key strings can consist of a single word (i.e., does not require at least two levels of type strings). The syntax of the metadata key is defined as follows:

```
      Metadata key := root-key-string *(“/” sub-key-string)
                *(“;” parameter)
                ; Matching of key-string
                ; is ALWAYS case-insensitive.
             ; There MUST ALWAYS be a root-key-string
```

```
root-key-string := token

sub-key-string := token

parameter := attribute "=" value

attribute := token
             ; Matching of attributes
             ; is ALWAYS case-insensitive.

Value := token / quoted-string

token := 1*<any (US-ASCII) CHAR except SPACE, CTLs,
           or tspecials>

tspecials :=  "(" / ")" / "<" / ">" / "@" /
              "," / ";" / ":" / "\" / <">
              "/" / "[" / "]" / "?" / "="
              ; Must be in quoted-string,
              ; to use within parameter values

quoted-string = <"> *(qtext/quoted-pair) <">; Regular qtext or
                                            ;   quoted chars.

Qtext       =  <any CHAR excepting <">,     ; => may be folded
               "\" & CR, and including
               linear-white-space>

quoted-pair =  "\" CHAR                     ; may quote any char

                                               octal    decimal
CHAR        =  <any ASCII character>        ; (  0-177,  0.-127.)
```

When the values are returned from a query the key will include some additional parameters providing further information about how to interpret the value. For example, the key returned from a query for author may look like:

```
author;valtype=whcar*
```

which indicates the value of the author string can be found in the wide character array member of the key-value pair structure. The valtype parameter will be included with every returned key since it is necessary to specify the member of the key-value pair structure to use.

| Key string | Description | Notes |
|---|---|---|
| album | Album name | Value is typically a null-terminated string (either narrow or wide character). |
| artist | Artist or performer | Value is typically a null-terminated string (either narrow or wide character). |
| author | Author or writer | Value is typically a null-terminated string (either |

| | | |
|---|---|---|
| | | narrow or wide character). |
| classification | Classification | Value is typically a null-terminated string (either narrow or wide character). |
| clip-type | High-level classification of clip describing whether it is local, streaming, download, etc. | Value is a null-terminated character string. Defined values include: local, streaming, download, fasttrack. |
| comment | Comment string | Value is typically a null-terminated string (either narrow or wide character). |
| compilation | Typically found in music content. | The value is a null-terminated string (either narrow or wide character). There is usually also a language code. |
| composer | Music composer | Value is typically a null-terminated string (either narrow or wide character). |
| copyright | Copyright holder | Value is typically a null-terminated string (either narrow or wide character). |
| date | Creation date | The date value will be returned as a string represented in a subset of ISO 8601 format. For example, “”yyyy”, “yyyy-mm”, etc where yyyy represents the year and mm the month. See the ID3 specification or reference [2] for other possible examples and more details. |
| description | Brief description of the content | Value is typically a null-terminated string (either narrow or wide character). |
| duration | Duration / length of the clip | Could be returned as an integer representing the duration along with a timescale or the string “unknown” if the duration is not known. See description below for more details. |
| duration-from-metadata | Duration of the clip provided in it’s metadata | Value is an unsigned 32-bit integer representing the duration along with a timescale. |
| genre | Genre | The value will typically be an integer code or string. See description below for details. |
| graphic | Location of an associated graphic or the actual graphic. | Value is typically a null-terminated string (either narrow or wide character) or an attached picture using a format like the ID3 attached picture format. |
| id | Product ID / SKU / unique ID | No specific standard defined for this field, so it would typically be returned as a string. |
| keyword | Content specific keyword(s) | The value is a null-terminated string (either narrow or wide character). There is usually also a language code. |
| lyricist | Lyricist. Typically found in music content. | The value is a null-terminated string (either narrow or wide character). There is usually also a language code. |
| lyrics | A simple string containing the words spoken or sung within the song. | The value is a null-terminated string (either narrow or wide character). There is usually also a language code. |
| music-selling-agency | Typically found in music content. | The value is a null-terminated string (either narrow or wide character). There is usually also a language code. |

| | | |
|---|---|---|
| music-label | Typically found in music content. | The value is a null-terminated string (either narrow or wide character). There is usually also a language code. |
| music-rights-holder | Typically found in music content. This could be different from copyright. | The value is a null-terminated string (either narrow or wide character). There is usually also a language code. |
| music-rights-information | Typically found in music content. | The value is a null-terminated string (either narrow or wide character). There is usually also a language code. |
| music-url | Typically found in music content. | The value is a null-terminated string (either narrow or wide character). |
| performer | Performer. Typically found in music content. | The value is a null-terminated string (either narrow or wide character). There is usually also a language code. |
| playlist | Playlist name | Value is typically a null-terminated string (either narrow or wide character). |
| podcast-url | Podcast URL | The value is a null-terminated string (either narrow or wide character). |
| purchase-date | Content purchase date | The date value will be returned as a string represented in a subset of ISO 8601 format. For example, “”yyyy”, “yyyy-mm”, etc where yyyy represents the year and mm the month. See the ID3 specification or reference [2] for other possible examples and more details. |
| rating | Rating information | There may be several different |
| src | Clip source / filename | Value is typically a null-terminated string (either narrow or wide character). |
| title | Title or name of the clip | Value is typically a null-terminated string (either narrow or wide character). |
| num-tracks | The number of tracks in the clip | Value is typically a 32-bit unsigned integer |
| version | Software version of the authoring software | Value is typically a null-terminated string (either narrow or wide character). |
| year | Year of recording/performance. Typically applies to music files. | Value is typically a 32-bit unsigned integer |
| popularimeter | Rating and playback counter for the music file | Value is typically a structure that has email, rating and playback counter. |

### 9.4.1 Limiting the Metadata Value Size

In certain cases it may be desirable to specify the maximum size of the metadata value that is being requested. This way the application can have some control over the amount of memory that will be used to return the metadata value. Since the metadata may include items like graphics (e.g., album art, etc), which can be fairly large, it is important for applications to have enough control to avoid out of memory conditions in memory-constrained situations.

The **maxsize** parameter in the request key string is used to specify the maximum size, in bytes, that should be returned for the requested value. It is an optional parameter that may be applied to any *variable-length* metadata value (i.e., strings, arrays, etc). The maxsize parameter does not apply to values that are returned as fixed-sized elements of the PvmiKvp union (e.g., int32, uint8, etc). The reason that it only applies to the variable-length values is that the PvmiKvp structure needs to be provided in every case to return the key value and report the required maximum size. Also the intention of the maxsize parameter is mainly to provide a way to deal with large metadata values. In case of metadata values that are strings, maxsize parameter will be interpreted to mean maximum size, in number of characters not including the NULL terminator, and not maximum size, in bytes.

With a maxsize parameter defined, there is the question of the behavior in the case that the maxsize is exceeded. Either the value could be returned truncated to the specified size or the information about the required size could be returned without the actual value (i.e., no space would be allocated to hold the value and no portion of the value be returned). Truncation is reasonable where an incomplete part of the value is potentially meaningful and useful (e.g., a string value). For cases where the value is really only useful when it can be returned in its entirety (e.g., a graphic image), then it does not make sense to truncate the value. Instead the request should be answered by indicating the required amount of memory to retrieve the complete value. Since the majority of the common metadata values are strings, the default behavior in the case where the value size exceeds the specified maxsize parameter will be to return the truncated value. However, an optional boolean parameter called ***truncate*** can be specified to indicate the desired truncation behavior. For example,

```
title;maxsize=100;truncate=false
```

indicates that the returned title value should have a max size of 100 bytes, and if the actual size exceeds that length, only the information about the required size should be returned (i.e., no truncation should happen). By contrast, the request

```
title;maxsize=100;truncate=true
```

differs by the fact that the title value should be truncated to at most 100 bytes. Note that the request with

```
title;maxsize=100
```

is equivalent to the one with the *truncate=true* parameter since truncation is the default behavior. ***Note that strings consisting of multibyte characters (e.g., UCS-2, UTF-8, etc) will be truncated to a whole number of characters that is less than or equal to the specified number of bytes.***

If the metadata value is larger than the specified maxsize, the required number of bytes is returned in the ***reqsize*** parameter. The required size is returned regardless of whether the value is truncated and returned or not. For example, if the request for the title is truncated at 100 bytes but the actual size of the string is 137, then the returned key string would look like:

```
title;valtype=wchar*;reqsize=137
```

The reqsize parameter is only included in cases where the entire value is not returned (i.e., either when it was truncated or no part of value is returned).

### 9.4.2 Duration

The duration value is typically returned as an integer value and may include an optional parameter that specifies a timescale. *If no timescale is specified, then the default is milliseconds*. Some examples include

`duration;valtype=uint32` (the duration is an integer representing milliseconds)
`duration;valtype=uint32;timescale=8000` (the duration is an integer representing the duration in a timescale of 8 kHz)
`duration;valtype=char*` (the duration is returned as the string “unknown”).

The duration is not stored explicitly in all the supported file formats. File formats like mpeg4 store the duration value explicitly so it is a simple and quick matter to extract the value. Other simpler file formats like mp3, aac, and amr consist of a concatenation of encoded frames without the duration explicitly stored anywhere. Therefore, the duration must be determined by parsing the entire file, which can be computationally expensive and time-consuming in these cases. By default, the duration will be returned as unknown in these cases initially and an event will be sent once the duration has been determined as a part of normal playback. However, it is possible to request that the duration be computed by including an additional option in request key. The form of the request key would look like:

```
duration;compute=true
```

to request that the duration is computed if necessary and possible.

The *duration-from-metadata* key is for situations where the duration is provided as an optional part of metadata and is not guaranteed or even necessarily reliable for the content type. For content that duration derived from metadata in a consistent and reliable way, the *duration* metadata key will be used. This can be queried via

```
duration-from-metadata;valtype=uint32
```

### 9.4.3 Genre

The genre value is often stored as an integer that corresponds to one of the values defined by ID3v1. In these situations, the returned value would be an integer and the returned key would include a qualifying parameter to indicate that it should be interpreted as an ID3v1 genre code as follows:

```
genre;valtype=uint32;format=id3v1
```

In the case of ID3v2, the classification may consist of a mixture of the ID3v1 code and an arbitrary string. In this case, the genre would be returned as a string with id3v2 indicated in the format as follows:

```
genre;valtype=wchar*;format=id3v2.
```

In some cases the genre may simply be a free-form string. In those cases the format parameter would not be provided because there is no special way of interpreting the value other than as a string. For example, it the key string would like the following:

```
genre;valtype=wchar*.
```

### 9.4.4 Graphic

The graphic value may be either a reference to an external image (i.e., stored separately from the media source) through a URL string or the image itself. In the case of the external reference, the information would be returned as a URL string. For example,

```
graphic;valtype=char*
```

is an example of a key string for a external reference graphic where the character string is a URL. If the graphic is returned as a character string with no other format specification, then it should be interpreted as a URL.

A popular format for directly holding the image within the media file is the ID3v2 attached picture format (i.e., the APIC frame). This same format is also used within ASF files for the WM/Picture metadata value. The format parameter for the key string indicates that the value is in the APIC format as follows:

```
graphic;format=APIC;valtype=ksv
```

The format includes the following information:

- A MIME type string describing the format of the image data
- A picture type code that classifies the content of the image
- A text description
- The binary picture data.

The picture type is one byte with values defined in the ID3v2 specification [3]. For convenience, the current table at the time this document was produced is included below, but the reference, [3], should serve as the official source of the defined values.

| Code (in hex) | Description | Code (in hex) | Description |
|---|---|---|---|
| $00 | Other | $0B | Composer |
| $01 | 32x32 pixels 'file icon' (PNG only) | $0C | Lyricist/text writer |
| $02 | Other file icon | $0D | Recording Location |
| $03 | Cover (front) | $0E | During recording |
| $04 | Cover (back) | $0F | During performance |
| $05 | Leaflet page | $10 | Movie/video screen capture |
| $06 | Media (e.g., label side of CD) | $11 | A bright coloured fish |
| $07 | Lead artist/lead performer/soloist | $12 | Illustration |
| $08 | Artist/performer | $13 | Band/artist logotype |
| $09 | Conductor | $14 | Publisher/Studio logotype |
| $0A | Band/Orchestra | | |

In general, there can be an arbitrary number of APIC frames associated with a file, but there may only be one instance of type $01 and one instance of type $02 according to the specification. The key string syntax for these

It may be desirable to request information on the number of APIC frames in the file before actually requesting them. This information can be requested by using the key string

```
graphic/num-frames;format=APIC
```

Since there can be multiple frames in the file it may be desirable to obtain the descriptions and select the frame(s) of interest before actually requesting image data. The following key string can be used to request the multiple entries.

```
graphic/description;format=APIC;index=X…Y
```

where the values X and Y refer to the start and end index values of the requested frames (in the range 0 to num-frames-1). The structure returned is the same as the one used for the full APIC information except the binary image data buffer is empty (i.e., the description string, text encoding, mime type, and picture type, and the size of the binary image data are returned). To request the full value including the binary image data, the request key string would simply use 'graphic' string as in this example:

```
graphic;format=APIC;index=X.
```

It is also possible to restrict the query to a specific picture type by adding the "pict-type" parameter to the key string. In that case, the returned values are narrowed to the set of the frames that have the matching picture type. For example, the key string

```
graphic/num-frames;format=APIC;pict-type=0F
```

specifies that the value returned should correspond to the number of frames with picture type equal to 0F. If the pict-type parameter is applied to a request with an index parameter, then the range of valid index values is restricted to lie between 0 and num-frames-1, where num-frames is the number of frames with that picture type. For example, to get the second graphic value with picture type 0F, the request might look like

```
graphic;format=APIC;pict-type=0F;index=1.
```

The maxsize parameter is another common key string parameter that can be applied to the graphic value request as described in Section 9.4.1. Refer to the API documentation for details of the structure that is returned for the APIC format.

All images by default are considered storable/savable. However there could be cases wherein the content provided might mark some of the images as not storable. In those cases a "not-storable" string would be added to the returned key string. For example key string for a non-storable image would look like

```
graphic;format=API;index=1;not-storable
```

## 9.5 Track-level Information

Certain file formats, such as mpeg4, as well as streamed presentations can contain multiple media streams or tracks. The track-level information provides details at the individual media level on such things as format, sample rate, bitrate, etc. The mechanism for accessing track information will apply for all clips regardless of how many tracks are included. For simple file formats, there will only be one track while others may include an arbitrary number. The metadata key, "num-tracks," will return the number of tracks within the clip. Information for individual tracks is accessed or qualified in the returned value with the "index" parameter, which has a range from 0 to (num-tracks – 1). For example:

```
track-info/type;index=0;valtype=char*
```

would be one possible key string for the track-level type information of the first track (i.e., index 0).

### 9.5.1 Compact Representation of Ranges

When querying for a list of available keys from a file source with multiple tracks, it will have a set of keys for track-level information where the only difference is the index parameter. One possible way of returning the set of keys is to simply include them all individually in the list (e.g., `track-info/type;index=0`, `track-info/type;index=1`, `track-info/type;index=2`, `track-info/type;index=n-1`). However, a more compact representation is allowed for the index using a format for expressing a range. For example, the string

```
track-info/type;index=0…8
```

represents a set of 9 keys, 0 through 8, in a single string using the range representation for the index.

### 9.5.2 General Information

The table below lists general track-level information that would be available for any track.

| Key string | Description |
|---|---|
| track-info/type | The type information for the media stream typically expressed as a MIME type. |
| track-info/track-id | The track-id is the identifier specified within the file if any. It may be different than the “index” parameter, which is simply used to iterate through the track-info metadata. |
| track-info/sample-rate | The sample-rate of the media in samples per second. Applicable to audio tracks. Provides the sampling rate of audio. |
| track-info/bit-rate | The bit-rate in bits-per-second. |
| track-info/duration | The track-level duration. The format is the same as the clip-level durations. |
| track-info/num-samples | The number of samples in the track. |
| track-info/selected | Boolean value that signals whether the track specified by the index is selected for playback or not. |
| track-info/frame-rate | Applicable only to video tracks. Provides an **approximate** estimate of the video frame rate. |
| track-info/codec-name | Value is typically a null-terminated string (either narrow or wide character). |
| track-info/codec-description | Value is typically a null-terminated string (either narrow or wide character). |
| track-info/codec-specific-info | The uint8 pointer provides the codec specific information. |
| track-info/track-number | The track-number is the identifier specified within the file if any. Typically found in music files and can be different from both “index” and “track-id” metadata fields. |
| track-info/max-bitrate | Maximum bit-rate in bits-per-second. |
| track-info/layer-id | Layer id for mp3 clips. |

### 9.5.3 Format Specific Information

Some track-level information is specific to the type of media. Below are the defined video and audio track-level information.

### Video-specific track-level information

| | |
|---|---|
| track-info/video/format | Detailed video format information (e.g., profile and level information for mpeg4) |
| track-info/video/height | Height of the video frame. |
| track-info/video/width | Width of the video frame. |
| track-info/video/display-height | Display height of the video frame. This need not be same as the decode height. |
| track-info/video/display-width | Display width of the video frame. This need not be same as the decode width. |

### Audio-specific track-level information

| | |
|---|---|
| track-info/audio/format | Detailed audio format information |
| track-info/audio/channels | Number of audio channels (e.g., 1 = mono, 2 = stereo, etc). |
| track-info/audio/bits-per-sample | Mainly relevant for PCM audio files. |

## 9.6 Codec Level Format Specific Information

The codecs may also expose similar types of information, which are actually extracted from the bitstream. The codec-level information can be more reliable than the track-level information at times (e.g., in some files the height and width information has been found to be incorrect). The format-specific codec-level information is shown below:

### Video-specific codec-level information

| | |
|---|---|
| codec-info/video/format | Detailed video format information (e.g., profile and level information for mpeg4) |
| codec-info/video/height | Height of the video frame. |
| codec-info/video/width | Width of the video frame. |

### Audio-specific codec-level information

| | |
|---|---|
| codec-info/audio/format | Detailed audio format information |
| codec-info/audio/channels | Number of audio channels (e.g., 1 = mono, 2 = stereo, etc). |
| codec-info/audio/sample-rate | The sample-rate of the audio data in samples per second. For PCM audio, it represents the frequency in hertz. |
| codec-info/audio/bits-per-sample | Bits-per-sample of the output PCM |

## 9.7 Language Codes

3GPP Release 6 defines a number of metadata elements as part of the asset information specified in the document TS 26.244v6.2.0. These metadata strings can be represented in different languages, so there is a language code associated with each entry to encode the language of the string. The language codes are stored as packed ISO-639-2/T language codes, which are basically 3 character codes assigned to each language. The table below lists a just a few examples of the languages and the associated language code, please refer to a reference on ISO-639-2/T for a complete list such [1]:

| 3-Letter Language Code | Description |
|---|---|
| eng | English |
| fre/fra | French |
| ger/deu | German |

If the language code exists it will be returned in the iso-639-2-lang parameter. Otherwise English should be assumed. It is expected that content may contain the same metadata in multiple languages, so the language parameter in the returned key string can be used to select the value in the appropriate language based on the user preferences. An example of a key string with a language code is

```
track-info/type;index=0;valtype=wchar*;iso-639-2-lang=ger
```

## 9.8 DRM Related Metadata

There are a number of metadata values related to license information for content protected with some form of digital rights management (DRM). For a particular piece of content, not all the values in the table will be available. This set of metadata provides information that describes the issuer of the license, which operations are allowed, when it expires, etc. Note that certain time-based licenses may have only start times, only expiration times, or both start and expiration times. Values will only be returned if the license has a corresponding value for that key string, so for example, if the license only has a start time then queries for the license-expiry would not return a value.

| Key string | Description | Notes |
|---|---|---|
| drm/is-protected | Absence of value indicates that the content is unprotected; however if value returned; indicates whether the content is DRM- protected (true) or DRM-unprotected (false). | Value is bool type when provided. |
| dla/license-issuer | This is the URI of the license issuer. | Value returned in the char* type. |
| dla/num-redirect | Number of license server re-directs that were followed in a direct license acquisition attempt. | Value returned in the uint32 type. |
| drm/allowed-usage | Provides information on the approved usage of the content. The returned value is a packed bit array with the possible permission values. | Packed bit array.<br>Possible values include:<br>Play, Pause, Resume, Seek forward, Seek backwards, stop, print, download, save, execute, preview. |
| drm/is-license-available | True/false value indicating whether the license is available. | Value returned in the bool type. |
| drm/auto-acquire | True/false value indicating whether there will be an attempt to automatically acquire a new license when necessary. | Value returned in the bool type. |
| drm/license-type | License types fall into following categories:<br>● time based (has an start and end time)<br>● duration based (a certain amount of time since first use)<br>● count based<br>● or a combination of count and one of time-related types | Value returned would be a string that will take any of<br>● the following forms:<br>● “time”, “duration”, “count”, “time-count”, “duration-count”, “unlimited” |

| | | |
|---|---|---|
| | • unlimited (no limit on the counted license) | |
| drm/num-counts | Counts remaining | Value is returned as a uint32 |
| drm/license-start | The starting time for the licensed interval | All start and end times are in ISO 8601 Timeformat<br>• The format is as follows. Exactly the components shown here must be present,<br>• with exactly this punctuation. Note that the "T" appears literally in the string to indicate the beginning of the time element, as specified in ISO 8601.<br>Year: YYYY (e.g., 1997)<br>Year and month: YYYY-MM (e.g., 1997-07)<br>Complete date: YYYY-MM-DD (e.g., 1997-07-16)<br>Complete date plus hours and minutes: YYYY-MM-DDThh:mmTZD (e.g., 1997-07-16T19:20+01:00)<br><br>Complete date plus hours, minutes and seconds: YYYY-MM-DDThh:mm:ssTZD (e.g., 1997-07-16T19:20:30+01:00)<br><br>Complete date plus hours, minutes, seconds and a decimal fraction of a second: YYYY-MM-DDThh:mm:ss.sTZD (e.g., 1997-07-16T19:20:30.45+01:00)<br>where: YYYY = four-digit year<br>MM = two-digit month (01=January, etc.)<br>DD = two-digit day of month (01 through 31)<br>hh = two digits of hour (00 through 23) (am/pm NOT allowed)<br>mm = two digits of minute (00 through 59)<br>ss = two digits of second (00 through 59)<br>s = one or more digits representing a decimal fraction of a second<br>TZD = time zone designator (Z or +hh:mm or –hh:mm) |

| | | |
|---|---|---|
| | | This profile defines two ways of handling time zone offsets:<br>● Times are expressed in UTC (Coordinated Universal Time), with a special UTC designator ("Z").<br>● Times are expressed in local time, together with a time zone offset in hours and minutes. A time zone offset of "+hh:mm" indicates that the date/time uses a local time zone which is "hh" hours and "mm" minutes ahead of UTC. A time zone offset of "-hh:mm" indicates that the date/time uses a local time zone which is "hh" hours and "mm" minutes behind UTC.<br><br>For example: 1994-11-05T08:15:30-05:00 corresponds to November 5, 1994, 8:15:30 am, US Eastern Standard Time. 1994-11-05T13:15:30Z corresponds to the same instant.<br><br>If the start time is not set then the value should be interpreted as "now". |
| drm/license-expiry | End time of licensed interval. | See previous description of the time format. A query for this key will not be answered if there is no specified end time. |
| drm/duration | Duration of the license specified in number of seconds. | The value will be returned as a uint32. |
| drm/license-store-time | The time when the license was added to the license store. | See previous description of the time format. A query for this key will not be answered if there is no support for looking up the time that the storage for a particular license store. |
| drm/is-forward-locked | True/false value indicating whether the content is forward locked. | Value returned in the bool type. |

## 4.13 Windows Media DRM

The following table defines which of the previously listed metadata fields are available for Windows Media DRM and provides additional specifics on the values.

| Key string | Notes/Availability |
|---|---|
| drm/is-protected | This key is always available. |
| drm/is-license-available | This key is always available. |
| dla/license-issuer | This key is available only after engine Init command has returned PVMFErrLicenseRequired. |
| dla/num-redirect | This key is available only after engine Init command has returned PVMFErrLicenseRequired. |
| drm/license-type | This key is available only when source intent includes GETMETADATA. This indicates the type of playback rights restriction that currently applies to the content. Specific values returned are:<br>•"unlimited": unlimited rights are available.<br>•"count": has a playback count restriction.<br>•"time": has an absolute start time and/or an end time restriction.<br>•"time-count": has a count restriction plus a start time and/or an end time.<br>•"duration": has a duration restriction. This is a duration relative to the first playback time, or in other words "duration after first use".<br>•"time-duration": has a duration after first use restriction combined with an absolute start time and/or an absolute end time. |
| drm/num-counts | This key is available only when source intent includes GETMETADATA and the current playback rights include a play count restriction. The returned value is the number of counts remaining. |
| drm/license-start | This key is available only when source intent includes GETMETADATA and the current playback rights include a start time restriction. See the previous table for the time value format. |
| drm/license-expiry | This key is available only when source intent includes GETMETADATA and the current playback rights include an end time restriction. See the previous table for the time value format. |
| drm/duration | This key is available only when source intent includes GETMETADATA and the current playback rights include an "expiration after first use" restriction. The data returned is the number of seconds that will remain after the first playback. After the first playback occurs, the reported rights will include the actual end time instead of the original "duration after first use" value. |
| drm/content-header | This key is available only after engine Init command has returned PVMFErrDrmLicenseNotFound or PVMFErrDrmLicenseExpired. The data returned is the DRM content header that can be used to acquire content licenses. |

## 9.9 Access to Other Metadata

Depending on the content format being accessed and the metadata storage scheme, there may be additional metadata entries that do not fall within the list of values described above. This situation is especially true for extensible metadata schemes like ID3v2. The parser used by the engine may not necessarily understand how to interpret the data in the metadata frame, but it can provide the raw data back to application for it to interpret. The form of the keystrings for requesting ID3v2 frames is:

```
id3v2/<four-character frame ID>
```

where <four-character frame ID> is the four-character code defined by the ID3 specification. If the key string is present, the returned value will include the ID3 version, the frame ID, frame size, frame flags, and the raw data contained in the frame. See the API documents for the exact definition of the id3v2 frame structure. This id3v2 frame structure will be returned in the key-specific value field of the returned key-value pair structure.

## 9.10 Receiving Metadata from Informational Event Callback

For server side playlist streaming sessions, PVPlayer engine also sends an unsolicited information event — PVMFInfoPlayListClipTransition. A playlist contains several playlist elements. When the client gets notified about the transition to a new playlist element, the player engine sends this event. It should NOT be used as an accurate indication of the transition point on UI because of the delay like jitter etc. This event also carries the extra meta data about the next playlist element. The event data is a PVMFRTSPClientEngineNodePlaylistInfoType struct:

```
typedef struct
{
 uint32 iPlaylistUrlLen;
 char *iPlaylistUrlPtr;
 uint32 iPlaylistIndex;
 uint32 iPlaylistOffsetSec;
 uint32 iPlaylistOffsetMillsec;

 uint32 iPlaylistNPTSec;
 uint32 iPlaylistNPTMillsec;

 //max 256
 uint32      iPlaylistMediaNameLen;
 char iPlaylistMediaNamePtr[256];

 //max 512
 uint32      iPlaylistUserDataLen;
 char iPlaylistUserDataPtr[512];
}PVMFRTSPClientEngineNodePlaylistInfoType;
```

## 9.11 Receiving Metadata during Clip Transition

For local playlist scenarios, PVPlayer sends certain metadata for each clip in the data source.

### 13.1.20 PVPlayerInfoClipInitialized

PVPlayer sends this event when playing from a clip list, to indicate that a new clip has been initialized and metadata is available. This event's local buffer contains the clip index.
The format of local buffer is as follows:

*Byte 1-4: uint32 (zero-based index of the initialized clip)*
*Byte 5-8: unused*

### 13.1.21 PVPlayerInfoClipPlaybackStarted

PVPlayer sends this event when playing from a clip list, to indicate that a new clip has playback has started. This event's local buffer contains the clip index.
The format of the local buffer is as follows:

*Byte 1-4: uint32 (zero-based index of the initialized clip)*
*Byte 5-8: unused*

### 13.1.22 PVPlayerInfoClipPlaybackEnded

PVPlayer sends this event when playing from a clip list, to indicate that a clip's playback has ended. This event's local buffer contains the clip index.
The format of the local buffer is as follows:

*Byte 1-4: uint32 (zero-based index of the initialized clip)*
*Byte 5-8: unused*

## 9.12 Metadata Retrieval Usage Example

To illustrate how metadata list is generated and returned to the user of PVPlayer SDK and how metadata value is returned, a sequence diagram between the user of PVPlayer SDK, PVPlayer engine, and two PVMF nodes that support metadata retrieval is shown below.

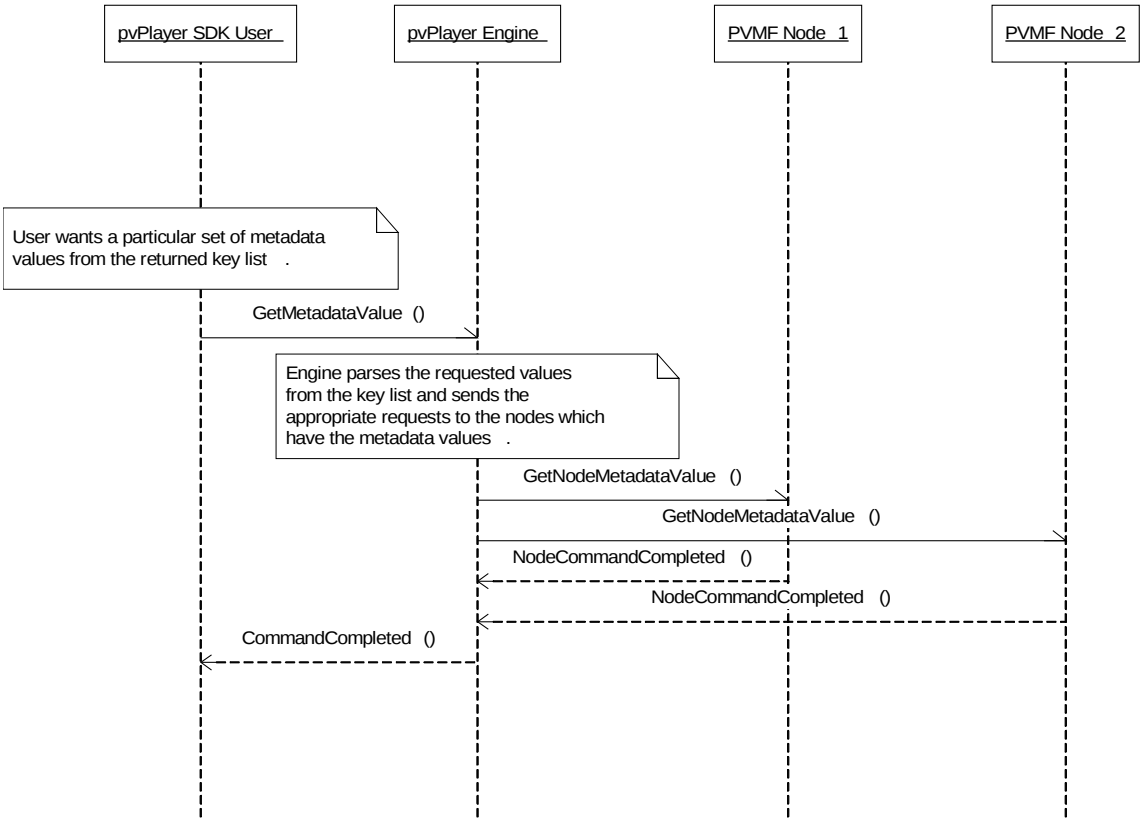



Figure 10: Metadata Retrieval Usage Sequence

## 9.13 Supported Key Strings in Select PVMF Nodes

The table below lists the supported metadata key strings in several PVMF nodes. The key string list is the comprehensive list, but actual key list could be a subset depending on the information available in the data source.

| PVMF Node | Supported Key Strings |
|---|---|
| PVMFMP4FFParserNode | author<br>title<br>description<br>rating<br>copyright<br>version<br>date<br>duration<br>num-tracks<br>track-info/type<br>track-info/track-id<br>track-info/duration<br>track-info/bit-rate<br>track-info/audio/format<br>track-info/video/format<br>track-info/video/width<br>track-info/video/height<br>track-info/sample-rate |
| PVMFMP3FFParserNode | title<br>artist<br>album<br>year<br>comment<br>copyright<br>genre<br>track-number<br>num-tracks<br>duration<br>track-info/bit-rate<br>track-info/sample-rate<br>track-info/audio/format<br>track-info/audio/channels<br>track-info/audio/layer-id |
| PVMFAACFFParserNode | title<br>artist<br>album<br>year<br>comment<br>copyright<br>genre<br>track-number<br>num-tracks<br>duration |

| | |
|---|---|
| | profile<br>channel<br>track-info/bit-rate<br>track-info/sample-rate<br>track-info/audio/format |
| PVMFAMRFFParserNode | duration<br>channel<br>num-tracks<br>track-info/bit-rate<br>track-info/audio/format |
| PVMFWAVFFParserNode | duration<br>num-tracks<br>track-info/bit-rate<br>track-info/sample-rate<br>track-info/audio/format<br>track-info/audio/channels<br>track-info/audio/bits-per-sample |
| PVMFVideoDecNode | codec-info/video/format<br>codec-info/video/width<br>codec-info/video/height |
| PVMFAVCDecNode | codec-info/video/format<br>codec-info/video/width<br>codec-info/video/height |
| PVMFAACDecNode | codec-info/audio/format<br>codec-info/audio/channels<br>codec-info/audio/sample-rate |
| PVMFWMADecNode | codec-info/audio/format<br>codec-info/audio/channels<br>codec-info/audio/sample-rate |
| PVMFWMVDecNode | codec-info/video/format<br>codec-info/video/width<br>codec-info/video/height |

# 10 Playback Position

PVPlayer engine provides the application with methods to obtain the current playback position of the media being played. The application can use the playback position data as strictly informational data to display to the user or to make decisions during media playback (e.g. pause playback 5 seconds into playback).

The position can be retrieved by having the application make API calls or PVPlayer engine can send the playback position periodically via the unsolicited informational event callback. For both methods, the application can change the playback position units from the default of millisecond time unit.

## 10.1 Retrieve Playback Position Using API Call

PVPlayer SDK provides two API to retrieve the current playback position from PVPlayer engine: GetCurrentPosition() and GetCurrentPositionSync(). Both APIs perform the same function but the latter completes the request synchronously instead of asynchronously. In both APIs, the user must provide a reference to a PVPPlaybackPosition object which is an input/output parameter. The input parameter portion is the iPosUnit field which allows the user to request the units to use for the playback position. The default units is in milliseconds but the user can request the position in other time units such as seconds, hours, and SMPTE time code, or non-time units such as percentage of whole clip, sample number, and offset from beginning of the file in bytes. Availability of playback position in non-time units would depend on the support from the underlying nodes and source media being used. If non-time units is not supported, these APIs will return with PVMFErrNotSupported error code.

## 10.2 Receive Playback Position from Informational Event

PVPlayer engine also sends the current playback position periodically as an unsolicited informational event with PVMFInfoPositionStatus event code and player specific event code of PVPlayerInfoPlaybackPositionStatus (=8193) in PVPlayerErrorInfoEventTypesUUID event code space (=0x46fca5ac, 0x5b57, 0x4cc2, 0x82, 0xc3, 0x03, 0x10, 0x60, 0xb7, 0xb5, 0x98). The position value is stored in the local data buffer of the informational event. The application is responsible for “listening” for this event in the informational event callback handler if it wants to obtain the current playback position by this method.

The position units and the time length of the reporting period can be queried and modified via the capability-and-configuration extension interface of PVPlayer engine. The default settings are milliseconds for playback position units and 1000 milliseconds for the reporting period. For more information on how to query and modify these settings via the capability-and-configuration interface, refer to the Capability Query and Configuring Settings section. Support for non-time position units would change based on the underlying nodes and source media being used. Therefore, if support for non-time position units becomes unavailable, PVPlayer engine will automatically change to the default of milliseconds.

# 11 Frame and Metadata Utility

A common use-case for player functionality involves retrieving the metadata information along with a frame from the video stream to be used as a thumbnail or other still image representation of the clip. For example there may be gallery view of the available content stored on the filesystem, which is presented to the user as a still image frame from each clip along with some metadata information such as title, author, etc. The player engine APIs can certainly be used directly to obtain the necessary information. However the PVFrameAndMetadataUtility simplifies the task for the application by hiding some of the interaction with the player engine for this use-case.

## 11.1 Creating and Deleting the Utility

Instances of the PVFrameAndMetadataUtility are created and deleted using static member functions of the factory class. The factory function used to produce a new instance of the utility class takes a MIME string argument, which specifies the desired output format for the video frame, as well as references to observer classes for receiving callbacks from the utility. Internally, the utility creates an instance of the player engine. The diagrams below show sequences for creating and deleting the utility instance.

The format of the video frame that will be returned in the GetFrame calls is specified as an argument to the factory function when creating an instance of the utility class. A MIME string is used to specify whether the frame should be YUV420, RGB16, etc. The header file pvmf_format_types.h contains a listing of many of the common MIME strings for the different video frame formats. If the output format cannot be supported for a given input source that is specified later, then an error will be returned from the GetFrame call.

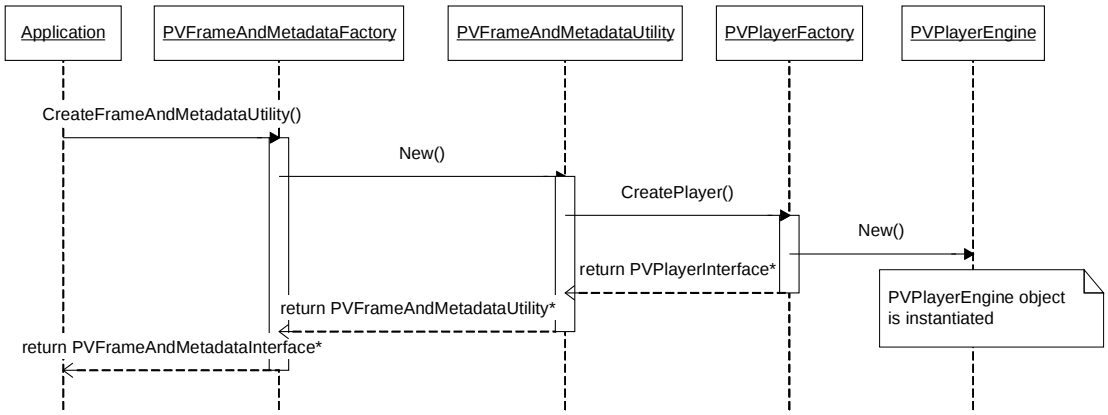



Figure 11: Create the Utility

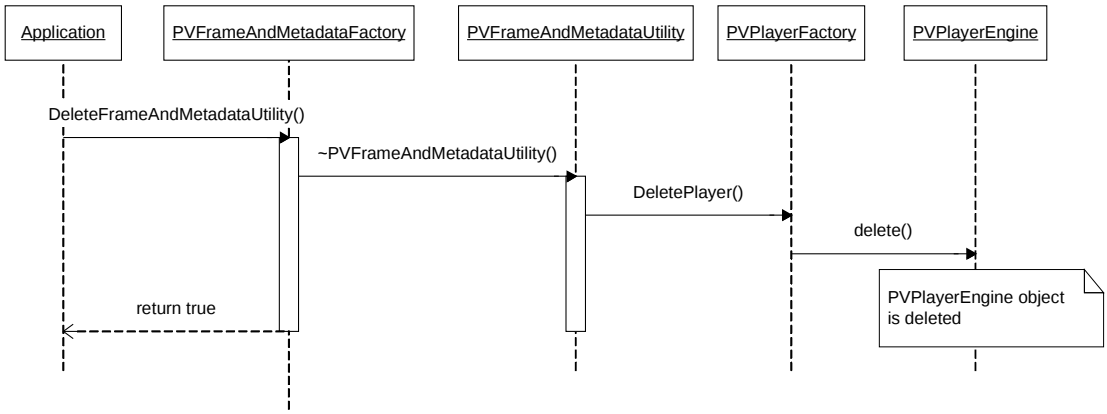



Figure 12: Delete the Utility

## 11.2 Options for Specifying the Desired Frame

The GetFrame() API is used to retrieve a frame specified in the frame selector argument. There are a few options for specifying the desired frame:

- the exact frame index with 0 corresponding to the first frame,
- the time offset of the frame,

These two options are used to select a specific frame based on either the frame index or the time offset of the frame. An example where this type of specification might be used is for creation of a thumbnail image from the first frame. The PVFrameSelector data type is used to hold the information on the desired frame.

In many cases, the first frame of the clip may not contain a meaningful image (e.g., the first frame may be a black frame). Therefore, another alternative is to let the Utility use an internal algorithm to autodetect a frame of interest. To achieve this, the user of the utility has to set the source context data with the BITMASK_PVMF_SOURCE_INTENT_THUMBNAILS intent. The following is an example of how it should be used.

```
// create the source context data for autodetection of thumbnails
iSourceContextData = new PVMFSourceContextData();
iSourceContextData->EnableCommonSourceContext();
//set the intent to thumbnails
 iSourceContextData->CommonData()->iIntent =
                        BITMASK_PVMF_SOURCE_INTENT_THUMBNAILS;

iDataSource->SetDataSourceContextData((OsclAny*)iSourceContextData);

iDataSource->SetDataSourceURL(wFileName);
iDataSource->SetDataSourceFormatType(iFileType);
OSCL_TRY(error, iCurrentCmdId=iFrameMetadataUtil->AddDataSource(*iDataSource,
                                                    (OsclAny*)&iContextObject));
```

## 11.3 Set Timeout for Frame Retrieval

The default timeout set for the frame retrieval is 30 seconds. The user of the utility has the option to alter the value of this timeout. This can be achieved by querying for the extension interface PvmiCapabilityAndConfig via the API QueryInterface(). The pointer to the interface obtained provides the flexibility to the user to set the timeout using the following KVP:

```
x-pvmf/fmu/timeout-frameretrieval-in-seconds;valtype=uint32
```

## 11.4 Usage Sequence

The main sequence for interfacing with the PVFrameAndMetadataUtility is shown in the figure below. As the diagram shows, the utility takes care of some of the steps of interaction with the player engine in order to get a specific frame or retrieve the metadata. The metadata is available to the application after the completion of the AddDataSource call to the utility. The AddDataSource, Init, AddDataSink, Prepare, Start, and Pause calls to the player engine are all hidden inside the processing of this request. The player engine is taken to a paused playback state to allow the datapath to be created and to allow the user to retrieve metadata from nodes within the datapath (e.g. codec information from decoder nodes).
There are two variants of the GetFrame call, which allow the frame buffer to either be provided by the application or the utility. The diagram below shows the case where the buffer is provided by the utility, in which case it must be returned once it is no longer needed using the ReturnBuffer call.

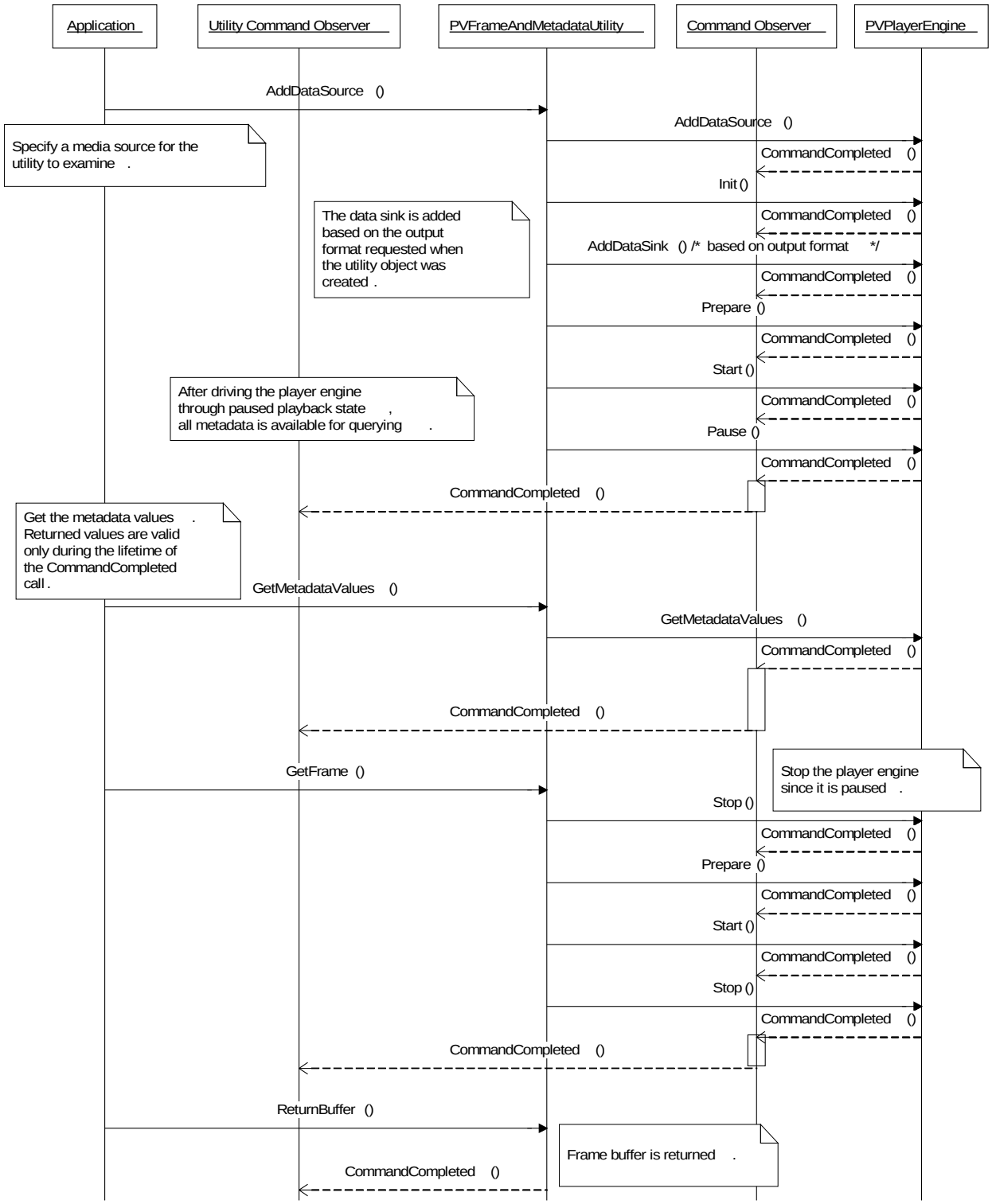



Figure 13: Frame and Metadata Utility Usage Sequence

# 12 Error and Fault Handling

## 12.1 Error Handling

Error is an erroneous system behavior that deviates from the design specifications. PVPlayer SDK will detect and handle any errors reported within its components or outside components (e.g. platform services, platform specific decoders). Based on the type of error, PVPlayer SDK will decide whether to report the error to the user of the SDK or not and whether to handle the error before continuing on. The reporting mechanism would depend on the interface between PVPlayer SDK and its user. With the OSCL-based interface, PVPlayer SDK reports errors via the command completion callback if the error occurs during an PVPlayer SDK API command processing or via the observer callback, PVErrorEventObserver, if the error is an unsolicited event.

The following section provides an overview of error types detected in PVPlayer SDK and error message reported by PVPlayer engine. Depending on the platform and PVPlayer SDK configuration, the list of error messages could be larger or smaller. For information on error events on a particular platform, refer to the PVPlayer SDK API document for that platform.

## 12.2 Error Codes

When PVPlayer engine reports an error, the error code would be one of PVMF status codes that provides a high-level description of the error. PVPlayer engine specific error code would be sent with the PVMF status code in the event extension interface pointer (PVInterface*) if available. The player engine specific error code would be encoded in the object pointed by the interface pointer and can be retrieved using PVMFErrorInfoMessageInterface extension interface methods.

Please refer the **pvmf_return_codes.pdf** document for the complete list of error status codes. PVPlayer engine specific error code would be in the range from 1024 to 8191 as specified by *PVPlayerErrorEventType* enum in *pv_player_interface.h*. The UUID for PVPlayer engine specific error code collection and event codes is defined as *PVPlayerErrorInfoEventTypesUUID*.

PVPlayer engine specific error codes are listed below.

| PVPlayer Engine Error Code | Error Description |
|---|---|
| PVPlayerEngineErrSourceInvalid | User provides an invalid data source for multimedia playback |
| PVPlayerEngineErrSourceInit | Error when initializing data source |
| PVPlayerEngineErrSource | General non-fatal error from the data source |
| PVPlayerEngineErrSourceFatal | General fatal error from the data source |
| PVPlayerEngineErrSourceNoMediaTrack | Data source contains no media track for playback |
| PVPlayerEngineErrSinkInvalid | User provides an invalid data sink for multimedia playback |
| PVPlayerEngineErrSinkInit | Error when initializing data sink |
| PVPlayerEngineErrSink | General non-fatal error from the data sink |
| PVPlayerEngineErrSinkFatal | General fatal error from the data sink |
| PVPlayerEngineErrNoSupportedTrack | No supported media track for playback was found |

| | |
|---|---|
| PVPlayerEngineErrDatapathInit | Error when initializing the datapath and its nodes |
| PVPlayerEngineErrDatapath | General non-fatal error from the datapath or its nodes |
| PVPlayerEngineErrDatapathFatal | General fatal error from the datapath or its nodes |
| PVPlayerEngineErrSourceMediaDataUnavailable | Data source ran out of media data |
| PVPlayerEngineErrSourceMediaData | General error in the data source's media data |
| PVPlayerEngineErrSinkMediaData | General error in the data sink's media data |
| PVPlayerEngineErrDatapathMediaData | General error in the datapath's or its nodes' media data |
| PVPlayerEngineErrSourceShutdown | Error when shutting down the data source |
| PVPlayerEngineErrSinkShutdown | Error when shutting down the data sink |
| PVPlayerEngineErrDatapathShutdown | Error when shutting down the datapath and its nodes |

## 12.3 Error Code Translation and Error Chain

When components below PVPlayer engine (i.e. PVMF nodes) report an error, PVPlayer engine receives and processes the error, and if the error needs to be reported to the user of PVPlayer SDK, the error as one of PVMF status code is passed up. A PVMF status code is passed up so the user can expect a limited set of error codes.

But for users of PVPlayer SDK that can handle more specific error information from PVPlayer engine and components below PVPlayer engine, the error from PVPlayer engine contains a linked list that shows the trail of error message from the originator of the error to the error message that was received by PVPlayer engine and the error message generated by PVPlayer engine. To understand all this error message information would require the user to have access to the context specific error codes used by the generator of error message at each level. To allow the context to be determined, the error list entry is based on PVMFErrorInfoMessageExtension, which provides access to the UUID for the error context with the error code. PVMFErrorInfoMessageExtension also allows the miscellaneous error message information (non-error code) to be retrieved as well via other extension interfaces. The user would need to understand the extension interface and know the UUID for the extension interface. Though PVMFErrorInfoMessageExtension is derived from PVInterface, it is not expected for the user of the SDK to keep a reference of the message even after the commandcomplete call back completes. So, the user of the SDK should NOT be increasing the reference counter of the message by doing a "addRef". For more information on PVMFErrorInfoMessageExtension including its features and usage, refer to its design document.

To illustrate how this error code translation is performed in PVPlayer SDK, an example with PVPlayer engine and data source nodes are shown below.

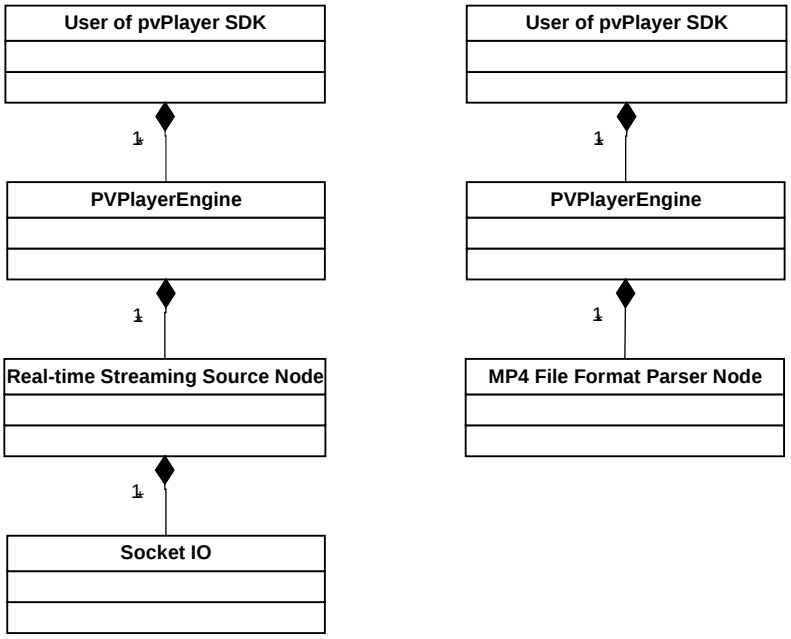



Figure 14: Class Diagram of Error Chain

The structure on the left shows the error propagation path for real-time streaming source when the error originates in network socket interface. The structure on the right shows the error propagation path for local file source when the error originates in the MP4 file format parser.

For the streaming case, the error originates in the socket IO level and real-time streaming source node receives a basic error message with an error code from socket IO. The streaming source node prepends its own error code to the socket IO error using PVMFErrorInfoMessageExtension's error chaining feature. Streaming source node packages its streaming specific error code with associated error details and sends an error event to PVPlayer engine. When PVPlayer engine receives the error event from the streaming source node, PVPlayer engine's own error code is prepended to the streaming source node's error and sends an error event to the PVPlayer SDK user. So the user of PVPlayer SDK receives PVMF error code and error messages from PVPlayer engine, streaming source node, and socket IO node which resulted in the engine level error. The streaming source node error message also provides more information about the error than just an error code by including RTSP error code and error strings if available. The diagram below shows how the error event, PVAsyncEvent, received by PVPlayer SDK would contain the error messages from PVPlayer engine, streaming source node, and socket IO node.

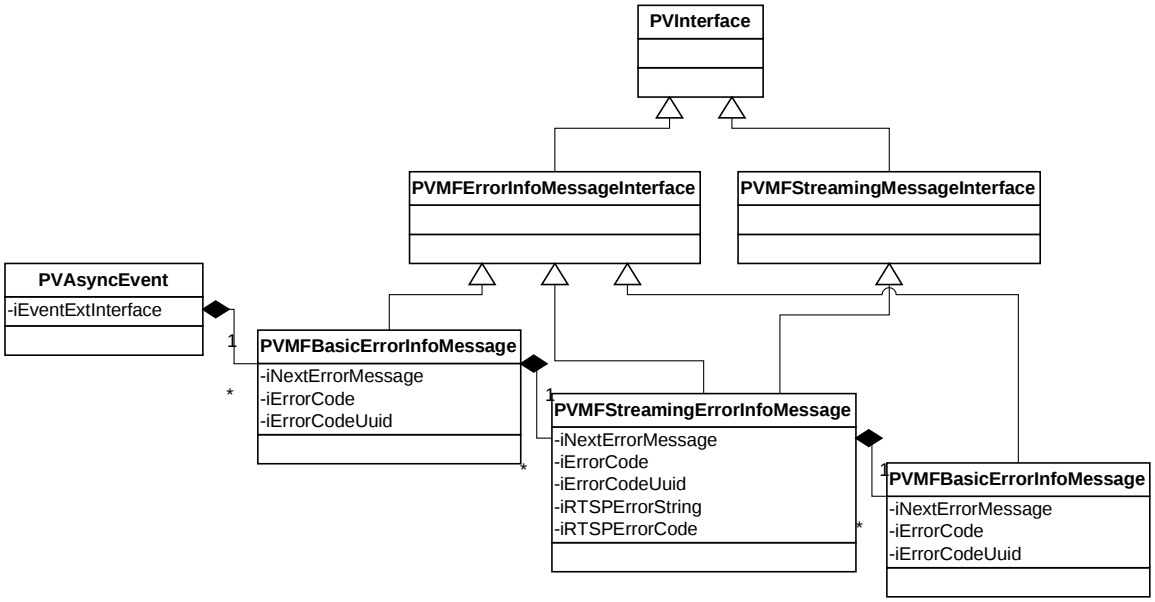



Figure 15: Streaming Error Event and Chain

For the local file case, the error originates in the MP4 file format parser node. In addition to the error code, the error message generated by the parser includes the MP4 atom where the error occurred. This error message is passed up to PVPlayer engine. PVPlayer engine then reports the error event to the PVPlayer SDK user with its own error code. The diagram below shows the error event and chain that the PVPlayer SDK user would receive for this case.

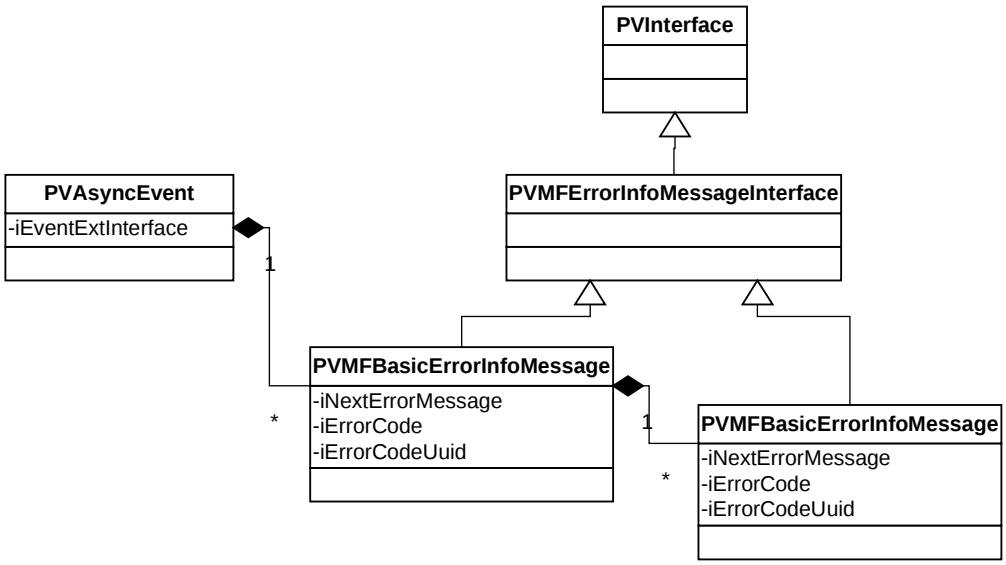



Figure 16: MP4 File Parsing Error Event and Chain

## 12.4 Typical Errors in Command Response

If a PVPlayer SDK API command fails, the failure is reported with error information in the CommandCompleted() callback. The following tables (one per API) list typical errors reported in response to API commands, the cause of the error, and expected handling by the user of PVPlayer SDK. As stated before, if an error is reported, the PVPlayer SDK user should check the PVPlayer engine state before performing any error handling of its own. Some errors might not be major or fatal and do not require any error handling.

**AddDataSource()**

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in idle state. Wrong state to call AddDataSource() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrArgument | Passed in player data source is invalid | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrNotSupported | Specified player data source (format type) is not supported in this PVPlayer SDK. | Command is rejected but engine does not go into error state. No error handling needed |
| PVMFErrNotSupported, PVPlayerErrSourceInit | Source node does not support the required extension interface | Check engine state to see if async error handling is occurring or not. Cannot use that particular data source.. |
| PVMFErrNoMemory | Required amount of memory not available in engine | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |
| PVMFErrNoMemory, PVPlayerErrSourceInit | Required amount of memory not available in source node. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |
| PVMFFailure | General failure code. Components are not behaving as expected. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |

**Init()**

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in idle state. Wrong state to call Init() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrNoMemory, PVPlayerErrSourceInit | Required amount of memory not available in source node. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |
| PVMFErrResource, PVPlayerErrSourceInit | Error while initializing the source (e.g. file parsing error, file corrupt). Check error | Check engine state to see if async error handling is occurring or not. Remove the data source if needed. |

| | | |
|---|---|---|
| | message for more specific info if available | |
| PVMFFailure | General failure code. Components are not behaving as expected. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |
| PVMFErrLicenseRequired, PVPlayerErrSourceInit | Authorization license needed to initialize the specified source | Should acquire a license (via player's license acquisition interface or other means) before calling Init() again. |
| PVMFErrAccessDenied, PVPlayerErrSourceInit | Rights management does not allow playback of the specified source. | Check engine state to see if async error handling is occurring or not. Remove the data source if needed. |

## AddDataSink()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in initialized state. Wrong state to call AddDataSink() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrArgument | Passed in player data sink is invalid | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrNotSupported | Specified player data sink (format type) is not supported in this PVPlayer SDK. | Command is rejected but engine does not go into error state. No error handling needed |
| PVMFErrNoMemory | Required amount of memory not available in engine | Command is rejected but engine does not go into error state. No error handling needed but should shutdown the engine. |

## Prepare()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in initialized state. Wrong state to call Prepare() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrNotReady | No player data sink added yet. | Command is rejected but engine does not go into error state. Add at least one valid player data sink before calling Prepare() again. |
| PVMFErrNotSupported | Previously specified player data sink is not supported | Check engine state to see if async error handling is occurring or not. Remove the unsupported data sink before calling Prepare() again. |
| PVMFErrNoMemory | Required amount of memory not available in engine | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |
| PVMFErrResourceConfiguration | Datapath could not created with specified source and | Command is rejected but engine does not go into error state. |

| | | |
|---|---|---|
| | sinks | Change the source and/or sinks. |
| PVMFErr…, PVPlayerErrSinkInit | Extension interface for file output sink node could be obtained | Check engine state to see if async error handling is occurring or not. Pass in a sink node instead of using the file output sink node. |
| PVMFErr…, PVPlayerErrSourceFatal | Source node reported a fatal error in response to one of the following node commands: Prepare, QueryDataSourcePosition, SetDataSourcePosition, or Start | Check engine state to see if async error handling is occurring or not. Should not playback this source. |
| PVMFErr…, PVPlayerErrDatapathInit | One datapath encountered error whiling initializing the datapath (e.g. setting up and connecting decoder and sink nodes) | Check engine state to see if async error handling is occurring or not. Should not continue. Check specific error codes for cause of error. |
| PVMFErr…, PVPlayerErrDatapathFatal | One datapath encountered error whiling starting data flow (i.e. calling node Start() on decoder and/or sink node). | Check engine state to see if async error handling is occurring or not. Should not continue. Check specific error codes for cause of error. |
| PVMFFailure | General failure code. Components are not behaving as expected. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |

### Start()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in prepared state. Wrong state to call Start() | Command is rejected but engine does not go into error state. No error handling needed. |

### SetPlaybackRange()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Wrong state to call SetPlaybackRange() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrArgument | Passed in reposition parameter is invalid (e.g. position value) | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrNotSupported | Specified reposition parameter is not supported. | Command is rejected but engine does not go into error state. No error handling needed |
| PVMFErr…, PVPlayerErrSourceFatal | Source node reported a fatal error in response to one of the following node commands: | Check engine state to see if async error handling is occurring or not. Check resulting state after error handling |

| | | |
|---|---|---|
| | QueryDataSourcePosition or SetDataSourcePosition | completes before continuing |
| PVMFErr…, PVPlayerErrSink | Sink node reported an error in response to SkipMediaData() command. | Repositioning during playback did not succeed properly but playback is still occurring. Stop playback first before continuing. |
| PVMFFailure | General failure code. Components are not behaving as expected. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |

## Pause()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in started state or already auto-paused due to source underflow. Wrong state to call Pause() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErr…, PVPlayerErrDatapathFatal | One datapath encountered error when pausing the datapath (e.g. node pause command failed on decoder and/or sink node). | Check engine state to see if async error handling is occurring or not. Check specific error codes for cause of error. Check resulting state after error handling completes before continuing |
| PVMFErr…, PVPlayerErrSourceFatal | Source node reported a fatal error in response to the node pause command | Check engine state to see if async error handling is occurring or not. Check resulting state after error handling completes before continuing |
| PVMFFailure | General failure code. Components are not behaving as expected. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |

## Resume()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in paused state or is in paused state due to auto-pause. Wrong state to call Resume() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErr…, PVPlayerErrDatapathFatal | One datapath encountered error when resuming the datapath (e.g. node start command failed on decoder and/or sink node). | Check engine state to see if async error handling is occurring or not. Check specific error codes for cause of error. Check resulting state after error handling completes before continuing |
| PVMFErr…, PVPlayerErrSourceFatal | Source node reported a fatal error in response to one of the following node command: QueryDataSourcePosition, SetDataSourcePosition, or Start | Check engine state to see if async error handling is occurring or not. Check resulting state after error handling completes before continuing |

| | | |
|---|---|---|
| PVMFErr…, PVPlayerErrSink | Sink node reported an error in response to SkipMediaData() command. | Repositioning when resuming did not succeed properly but playback has resumed. Stop playback first before continuing. |
| PVMFFailure | General failure code. Components are not behaving as expected. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |

## Stop()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in started, paused, or prepared state. Wrong state to call Stop() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErr…, PVPlayerErrDatapathFatal | One datapath encountered error when stopping the datapath (e.g. node stop command failed on decoder and/or sink node). | Check engine state to see if async error handling is occurring or not. Check specific error codes for cause of error. Check resulting state after error handling completes before continuing |
| PVMFErr…, PVPlayerErrDatapathShutdown | One datapath encountered error when tearing down and resetting the datapath. | Check engine state to see if async error handling is occurring or not. Check specific error codes for cause of error. Check resulting state after error handling completes before continuing |
| PVMFErr…, PVPlayerErrSourceFatal | Source node reported a fatal error in response to the node stop command | Check engine state to see if async error handling is occurring or not. Check resulting state after error handling completes before continuing |
| PVMFFailure | General failure code. Components are not behaving as expected. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |

## CancelAllCommands()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrArgument | The command is called during execution of AcquireLicense or CancelAcquireLicense | Command is rejected but engine does not go into error state. No error handling needed. |

## CancelCommand()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrArgument | If command is called to cancel AcquireLicense or | Command is rejected but |

| | | |
|---|---|---|
| | CancelAcquireLicense when they are being processed. OR If there is no Command to be cancelled. | engine does not go into error state. No error handling needed. |

### RemoveDataSink()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in initialized state. Wrong state to call RemoveDataSink() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrArgument | Passed in data sink is invalid | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFFailure | Specified data sink does not match an existing datapath. | Command is rejected but engine does not go into error state. No error handling needed. |

### RemoveDataSource()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Not in idle state. Wrong state to call RemoveDataSource() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrArgument | Passed in player data source is invalid | Command is rejected but engine does not go into error state. No error handling needed. |

### GetMetadataValues()

| Error code | Likely Cause | Expected Handling |
|---|---|---|
| PVMFErrInvalidState | Wrong state to call GetMetadataValues() | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrArgument | One or more passed-in parameter is invalid or there are no nodes with metadata interface | Command is rejected but engine does not go into error state. No error handling needed. |
| PVMFErrNoMemory | Required amount of memory not available in engine | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |
| PVMFFailure | General failure code. Components are not behaving as expected. | Check engine state to see if async error handling is occurring or not. Should not continue and should shutdown the engine. |

## 12.5 Typical Error Events

PVPlayer SDK errors that are not encountered when processing a PVPlayer SDK API command (e.g. error during playback) will be reported via the PVErrorEventObserver as an unsolicited event. The following table lists typical error events reported, the cause of the error, and expected handling by the user of PVPlayer SDK. As stated before, if an error is reported, the PVPlayer SDK user should check the PVPlayer engine state before performing any error handling of its own. Some errors might not be major or fatal and do not require any error handling. Please refer to section 5.1 of pvmf_return_codes.pdf document for the PVPlayerEngine error and extension codes with their likely causes.

## 12.6 Fault Detection, Handling and Recovery

Fault is an incorrect and unexpected system state. PVPlayer SDK will try to detect faults based on the information available. If the fault is avoidable or recoverable, PVPlayer engine will report the fault as an informational event and try to continue operation. If the fault is unrecoverable, PVPlayer engine will go into an error state and report the error to the layer above. The layer above will then need to reset or destroy the PVPlayer engine instance to resume operations from the fault. In some faults like memory allocation failure, PVPlayer engine will allow the leave to propagate up to the layer above to be trapped unless the PVPlayer SDK requirement for the platform does not allow leaving.

# 13 Usage Scenarios

To illustrate how PVPlayer SDK would be typically used, this section will present several PVPlayer SDK usage scenarios. Scenarios will cover different sources, playback features, and error conditions. The PVPlayer SDK represented in the scenarios will support all the features, but the interface would be the base level OSCL-based interface and all underlying nodes are OSCL-based software nodes. The lifelines in the sequence diagram will be limited to PVPlayer SDK interface and the user of the SDK unless the scenario calls for other object lifelines.

## 13.1 Instantiating PVPlayer SDK

The sequence diagram shows how the PVPlayer engine object is created via the factory component. After instantiation, PVPlayer engine is in IDLE state.

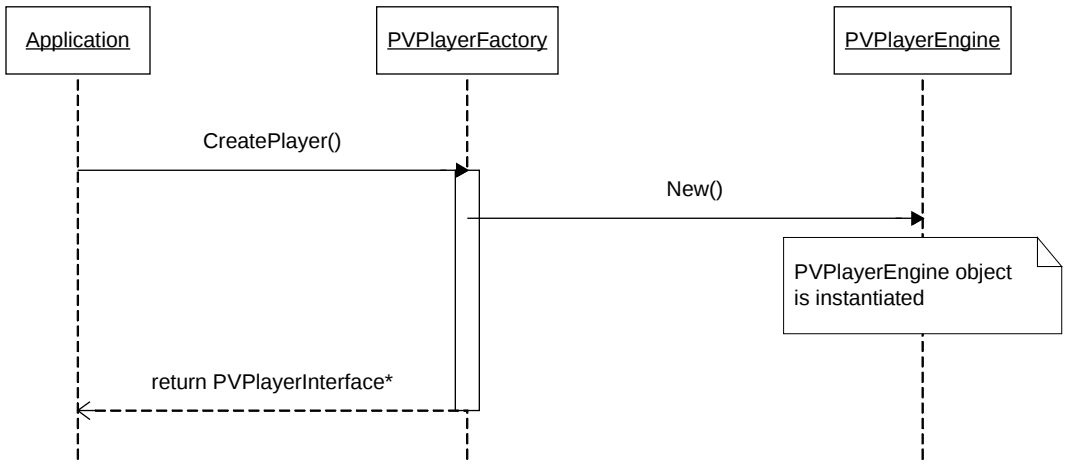



Figure 17: Sequence Diagram for Creating PVPlayer

## 13.2 Shutting down PVPlayer SDK

The sequence diagram shows how the PVPlayer engine object is destroyed via the factory component. PVPlayer engine object should be in IDLE state to properly destroy it.

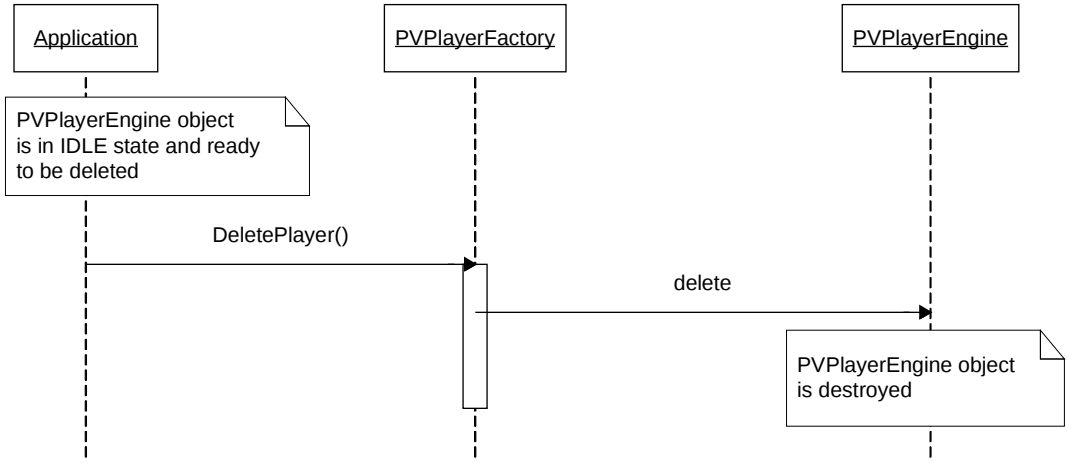



Figure 18: Sequence Diagram for Deleting PVPlayer

## 13.3 Open a Local MP4 File, Play and Stop

In this scenario, a local MP4 file containing audio and video tracks is specified as the data source, audio and video data sinks are added, playback is started, and then stopped after some time.

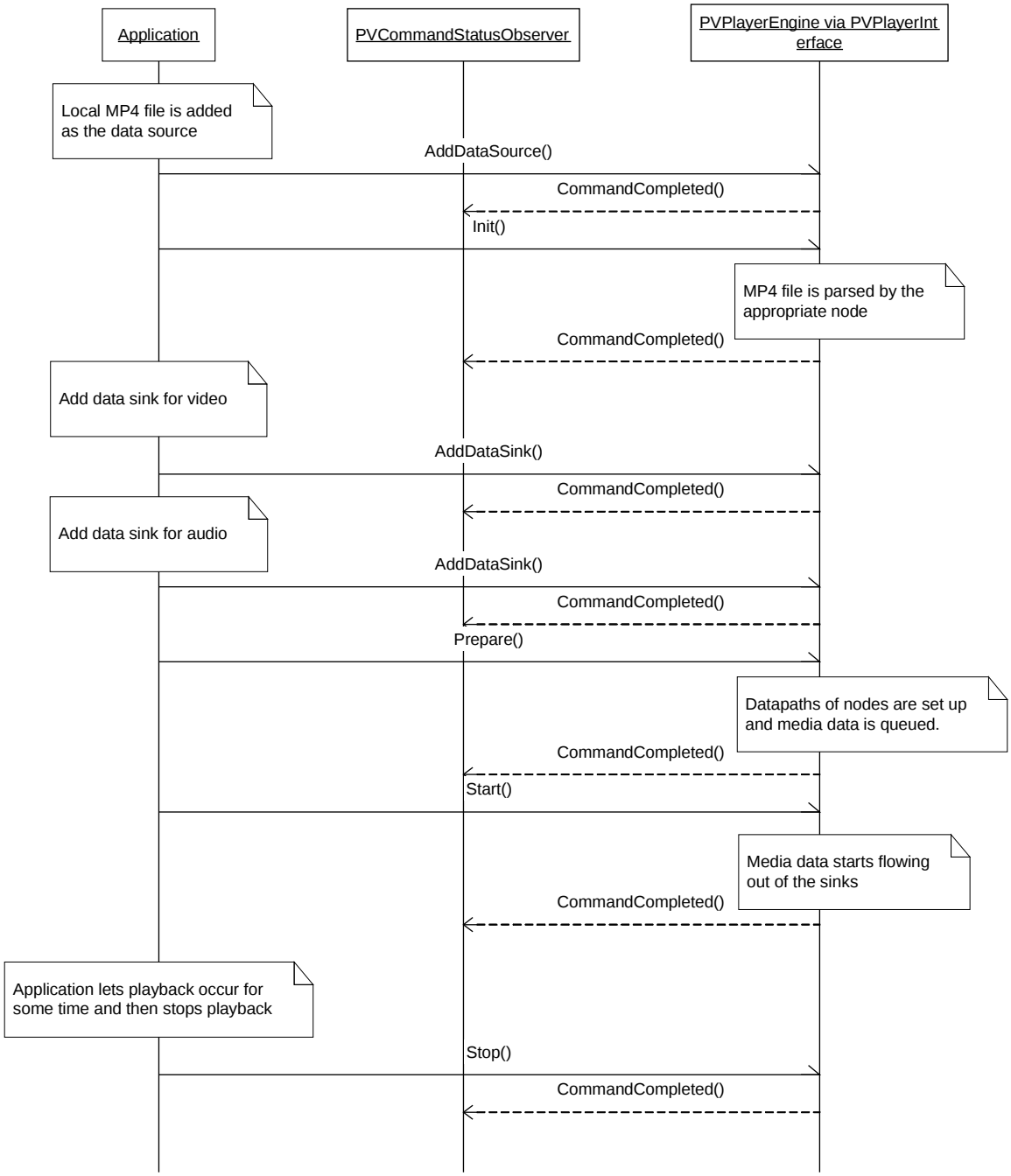



Figure 19: Open a Local MP4 File, Play and Stop

# 13.4 Open a RTSP URL, Play and Stop

In this scenario, a streaming source containing audio and video media tracks is specified by a RTSP URL. Then an audio and a video data sinks are added, playback started and then stopped after some time.

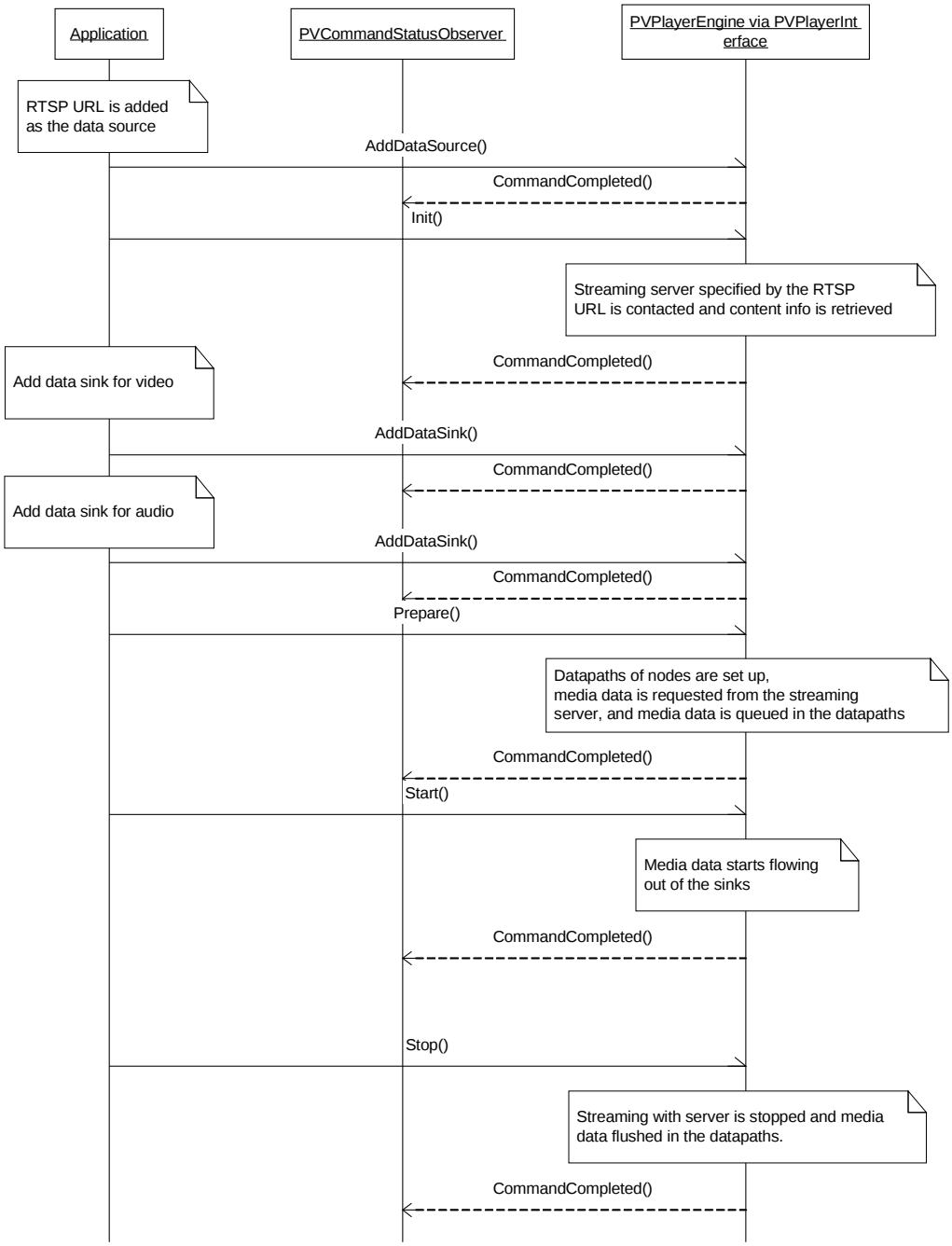



Figure 20: Open a RTSP URL, Play and Stop

## 13.5 Play a Local File Until End of Clip

This scenario is similar to a prior local file scenario but instead of ending playback due to the user calling a control API, the playback pauses since the end of clip is reached. The user still needs to call Stop() after the playback is automatically paused to stop the playback.

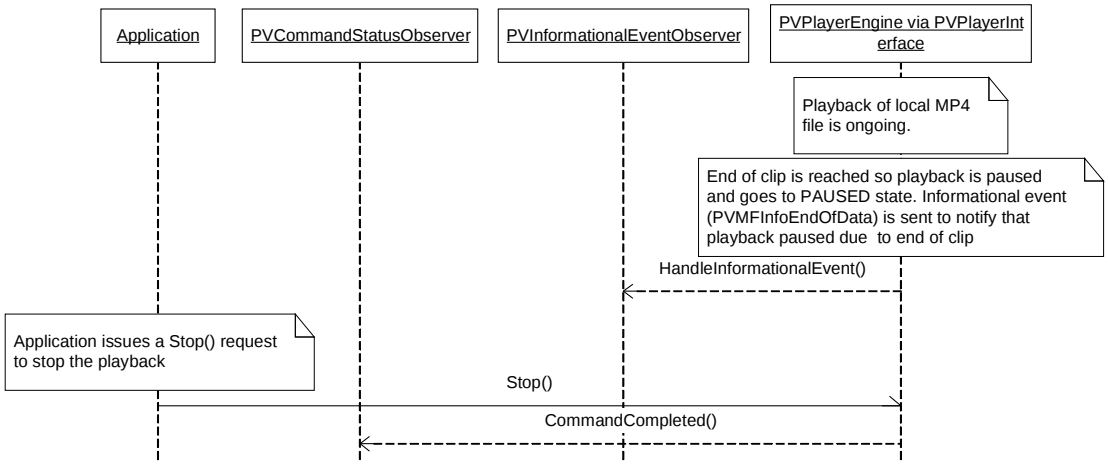



Figure 21: Play a Local File Until End of Clip

## 13.6 Play a Local File, Stop and Play Again

This scenario shows how to re-play a clip after it has been played and then stopped. PVPlayer engine goes to INITIALIZED state after Stop() command completes so Prepare() and Start() are called again to restart playback.

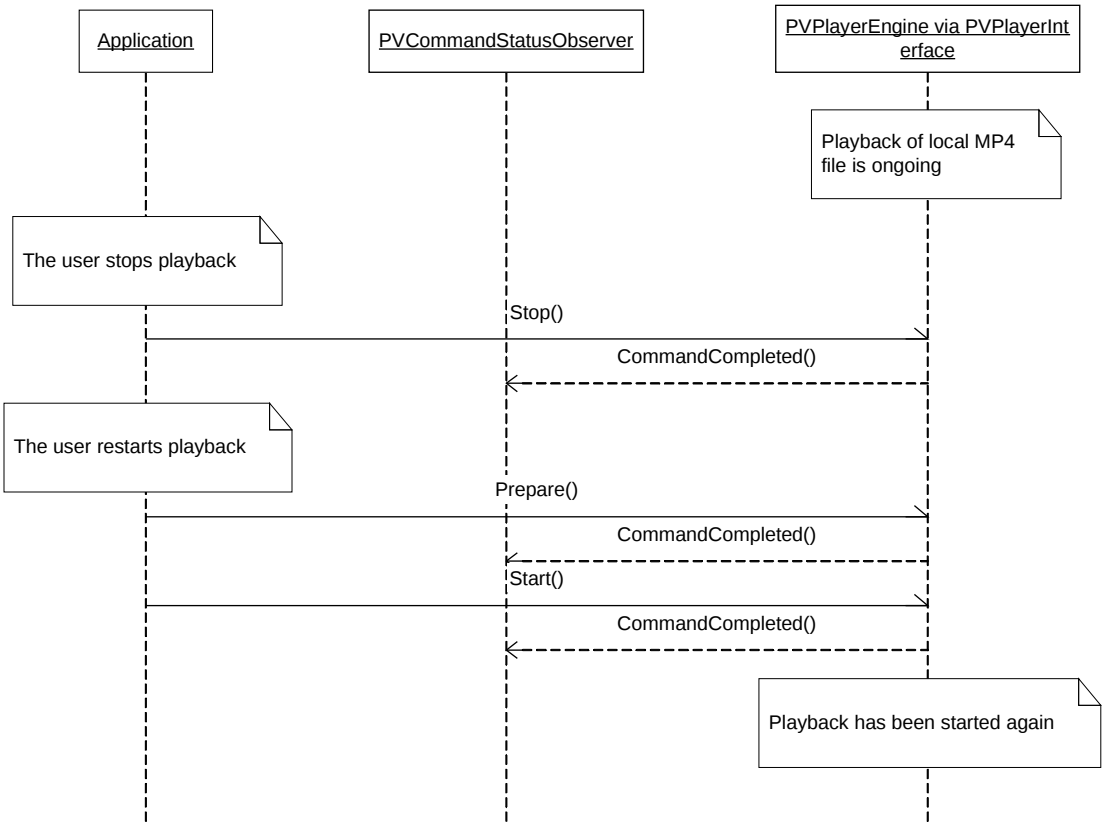



Figure 22: Play a Local File, Stop and Play Again

## 13.7 Play a local file, stop, open another file, and play

This scenario shows how to open another clip for playback after playing the current clip. The proper sequence for closing the current clip is shown.

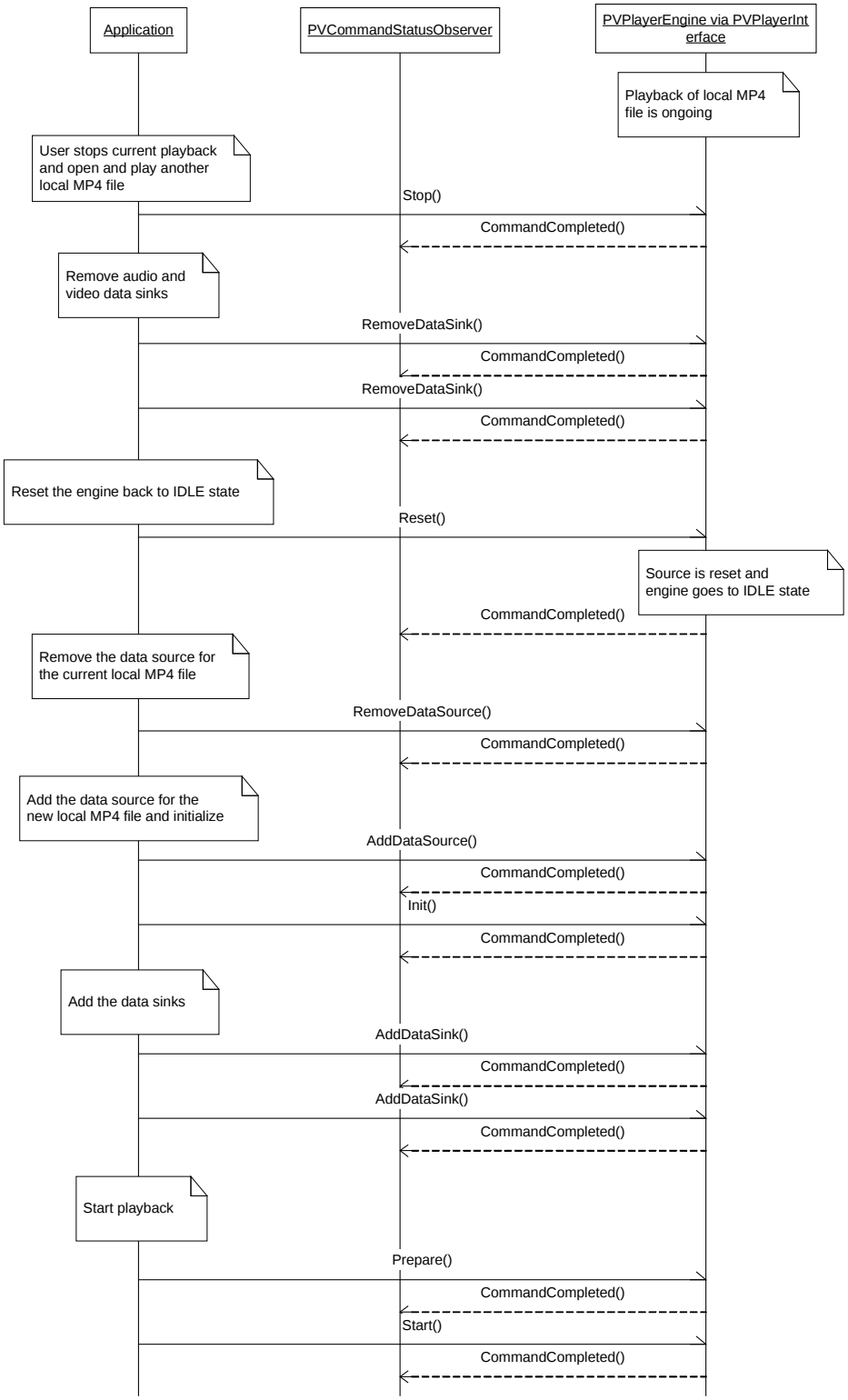



Figure 23: Play a local file, stop, open another file, and play

## 13.8 Play a local file, pause, and resume

In this scenario, a playback is paused and then playback is resumed.

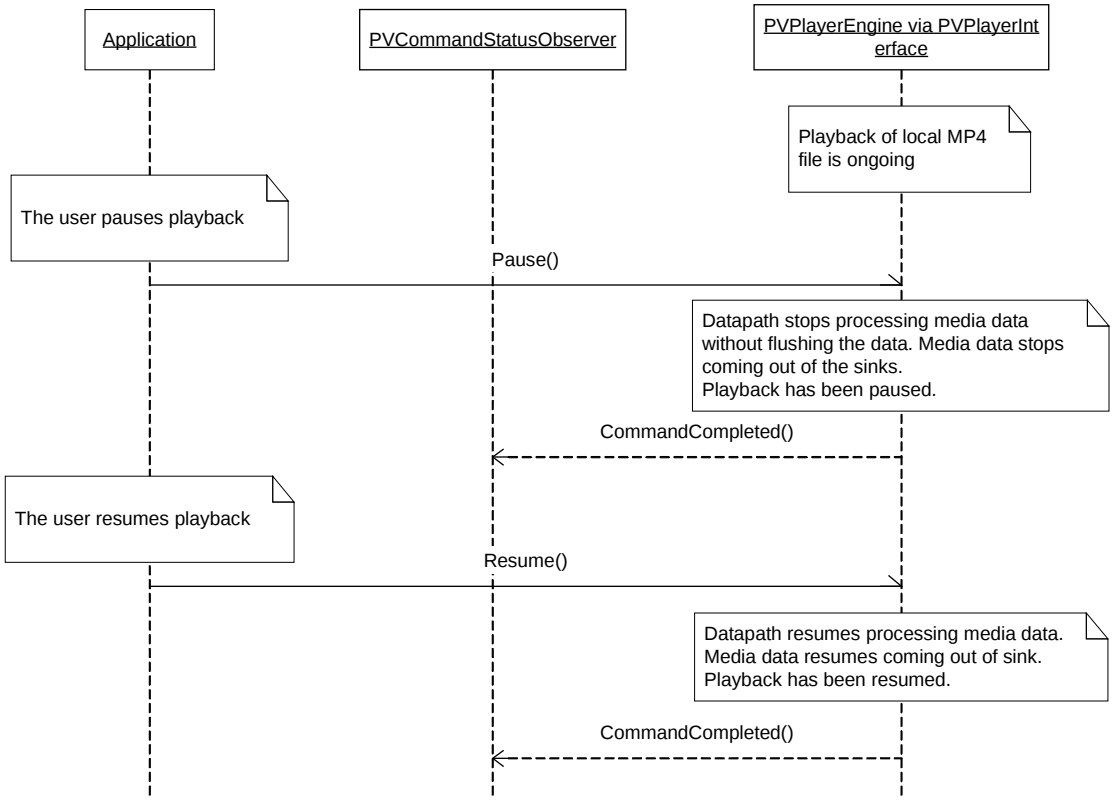



Figure 24: Play a local file, pause, and resume

## 13.9 Play a local file, pause, and stop

In this scenario, a playback is paused and then stopped when paused.

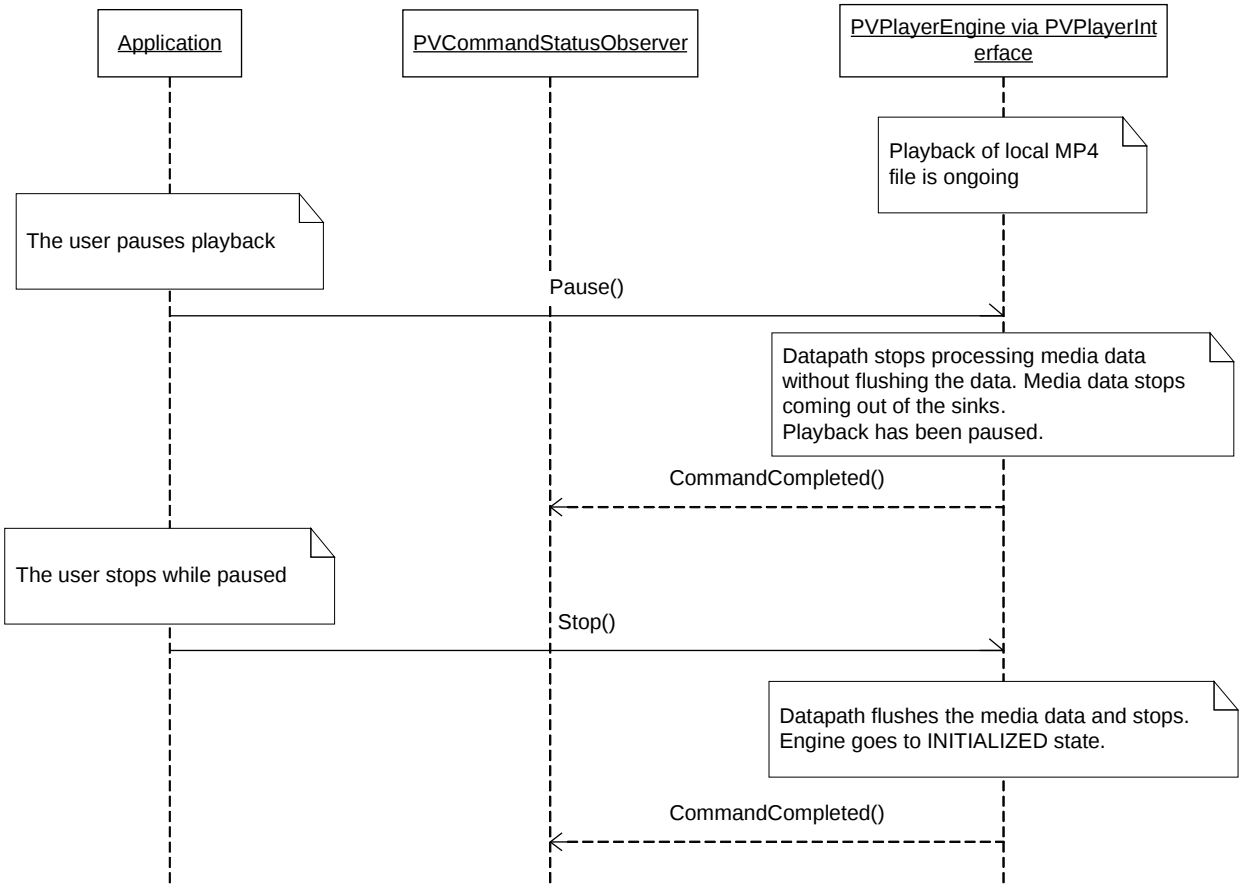



Figure 25: Play a local file, pause, and stop

## 13.10 Playback of DRM Protected Contents

When playing back DRM-protected content, the interaction between the application and player engine is very similar to playback of non-protected content with the exception of the handling of license acquisition and some of the events associated with license acquisition. If the license for a piece of content is available and valid, then playback will happen automatically using the same sequence of steps as with non-protected content. If no valid license is available, then the behavior depends on whether this piece of content had a previous valid license that expired. In most cases the engine simply notifies the application that acquisition of a valid license is required and relies on the application to decide, possibly through user interaction, if that should be done.

However, in some cases, it may be desirable to have the player engine automatically attempt to acquire the new valid license if the existing one has expired. An example would be some form of subscription service that the user has joined. In that case, the process is streamlined by having the player engine automatically attempt to acquire the license when necessary. A property stored along with the license determines whether it should be automatically acquired, so this behavior is only relevant after a license for a piece of content is acquired for the first time. The following subsections describe the interaction between the application and player engine for the different scenarios of handling DRM content.

### 13.10.1 Preparation to Play DRM Protected Contents

Before any DRM content can be played, the appropriate CPM (Content Policy Manager) plug-in modules must be registered for usage by player engine. For information on CPM plug-ins, refer to the PV CPM Plug-in Programming Guide.

CPM plug-ins are registered with a factory function and a MIME string, using the PVMFCPMPluginFactoryRegistryClient class. For each plug-in, the application instantiates the factory and passes the factory plus the MIME string to the registry. During playback of protected content, the player engine will check the registry and instantiate the plug-ins using their factory functions as needed. In applications that support multiple types of DRM, there will be multiple CPM plug-in modules. In a multi-DRM scenario, the application can register all available plug-ins without concern for which plug-in will be used for a particular piece of content, since player engine makes the determination during playback.

The CPM plug-ins can be registered anytime before playback of protected content occurs. The plug-ins will remain registered until the registry client session is closed. When the client session is closed, all plug-ins registered during that session are automatically cleaned up and removed. An application could register all plug-ins before creating player engine then cleanup plug-ins after player engine is destroyed.

Besides registering plug-ins, the application needs to set a Boolean flag in the local source data to tell player engine that the content may be DRM protected. In some cases, the application may not know whether a particular piece of content is protected or not. If there is any possibility that it is protected, the Boolean flag should be set to true.

The sequence diagram below shows the use of the CPM plug-in factory registry to register CPM plug-ins for use by player engine.

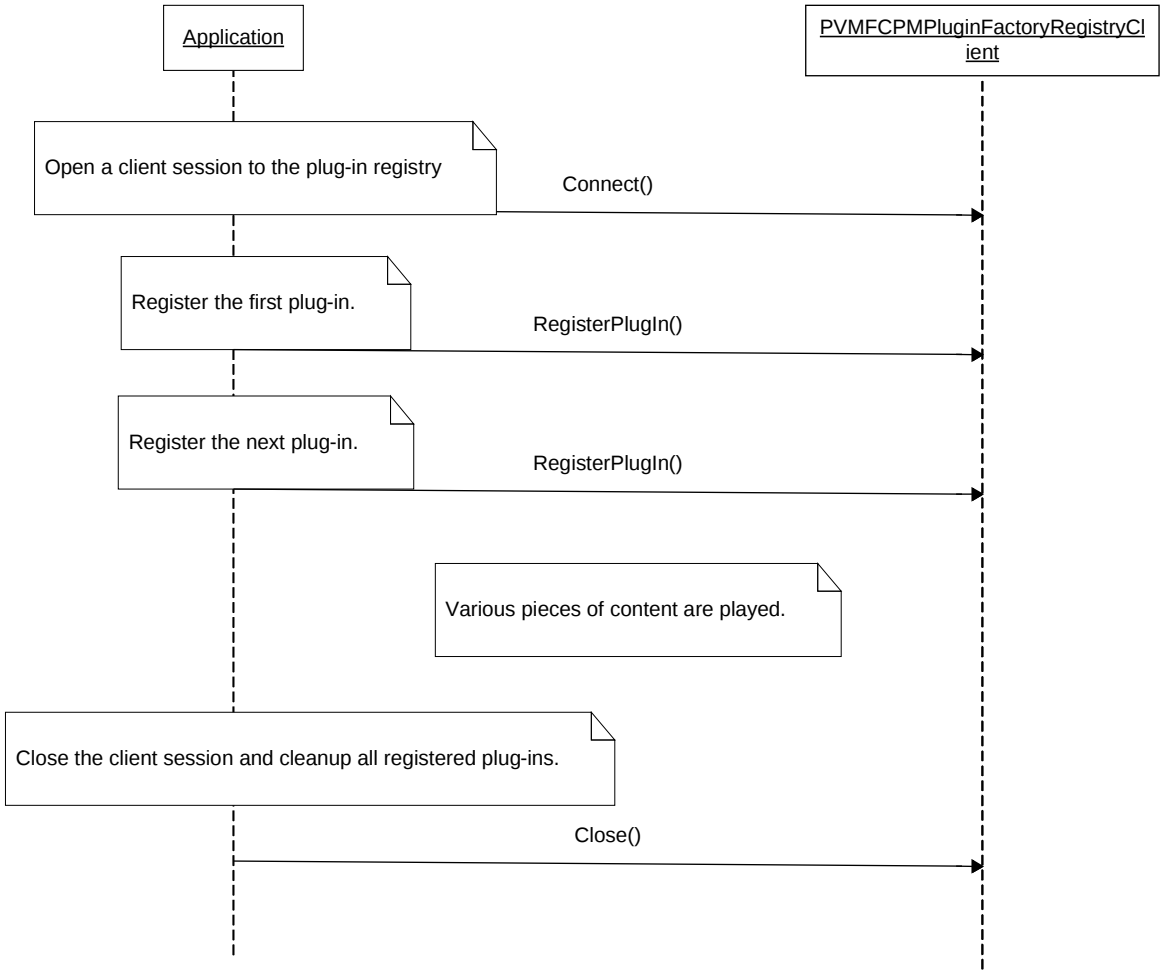



Figure 26: Preparation Sequence to Play DRM Protected Contents

## 13.10.2 Playback of DRM Content with a Valid License Available

Playback of DRM content with a license available is very similar to the usual playback sequence. The player engine will obtain the license from the license store during the Init phase and report success from the Init command if the license is valid. At that point the usual playback sequence can happen just as with non-protected content. If there is not license for this content in the license store or the existing license is invalid (e.g., expired), then the player engine will report an error and the sequence will proceed as described in Section 13.10.3. The sequence diagram below shows the details of the interaction between the application and player engine for the case of an available valid license.

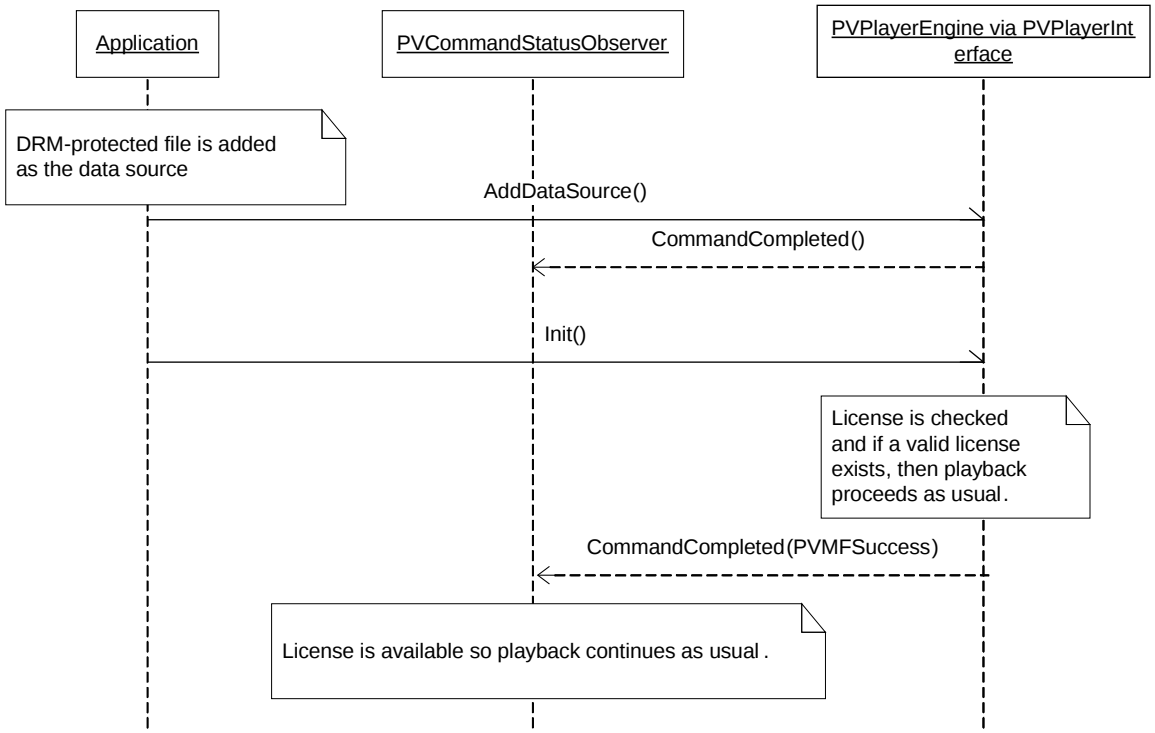


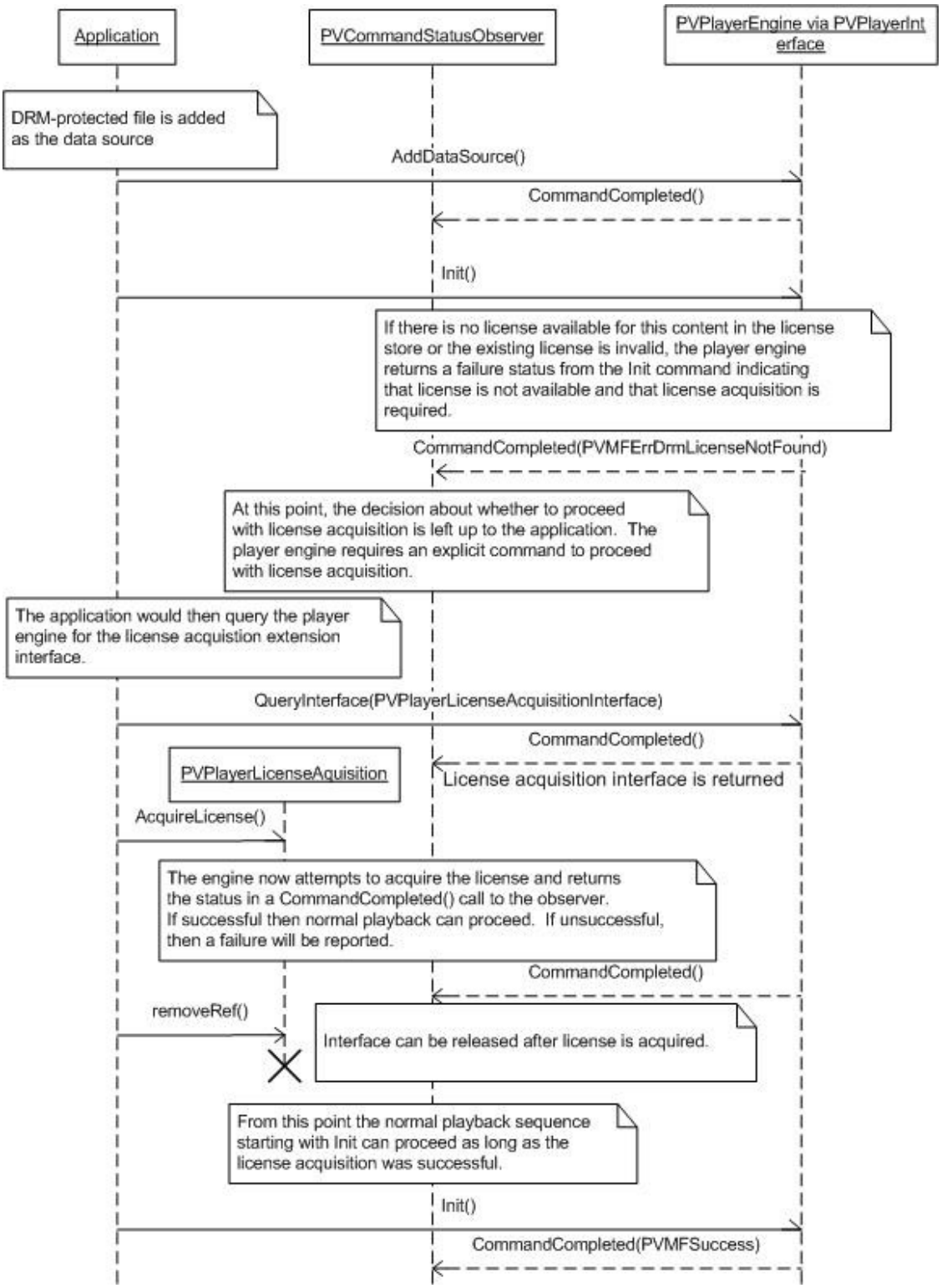

Figure 27: Playback of DRM Content with a Valid License Available

### 13.10.3 Playback of DRM Content *without* a Valid License Available

If there is no license at all for this content in the license store or the existing license is not valid, then the application will need to send the engine an explicit request to attempt to acquire the license. The call to Init will fail with the return code PVMFErrDrmLicenseNotFound or PVMFErrDrmLicenseExpired indicating that a license must be obtained before the clip can be played. At this point, the application can fail the playback and end the session with the player engine, acquire the license through some external process (i.e., outside the scope of the player engine), or request that the engine attempt to acquire the license.

In order to request that the engine acquire the license, the application must first get access to the appropriate extension interface of the player engine. This process is same as for any extension interface to engine, the QueryInterface is called with the relevant interface ID. The extension interface is returned in the CommandCompleted call to the observer. Once the application has the license acquisition interface, it can make the call to AcquireLicense. The engine attempts to get the license and returns the result in the CommandCompleted callback to the observer. Assuming the license acquisition was successful, the application can proceed with Init call again followed by the usual sequence for playback. In case there is a need for the application/user to be registered to a service, a PVMFErrDrmDomainRequired or PVMFErrDrmDomainRenewRequired error is returned. The application, in such a scenario, can make a call to JoinDomain. The necessary parameters for this call can be obtained from the license status. Assuming the registration was successful, the player engine also automatically attempts to re-acquire the license. The application can then proceed with Init call again followed by the usual sequence for playback. Once the application has finished with the license

acquisition interface, it should free the resource by calling the interface's removeRef() method. The sequence is shown in the diagrams below.


Application
PVCommandStatusObserver
PVPlayerEngine via PVPlayerInterface
DRM-protected file is added as the data source
AddDataSource()
CommandCompleted()
Init()
If there is no license available for this content in the license store or the existing license is invalid, the player engine returns a failure status from the Init command indicating that license is not available and that license acquisition is required.
CommandCompleted(PVMFErrDrmLicenseNotFound)
At this point, the decision about whether to proceed with license acquisition is left up to the application. The player engine requires an explicit command to proceed with license acquisition.
The application would then query the player engine for the license acquistion extension interface.
QueryInterface(PVPlayerLicenseAcquisitionInterface)
CommandCompleted()
PVPlayerLicenseAquisition
License acquisition interface is returned
AcquireLicense()
The engine now attempts to acquire the license and returns the status in a CommandCompleted() call to the observer. If successful then normal playback can proceed. If unsuccessful, then a failure will be reported.
CommandCompleted()
removeRef()
Interface can be released after license is acquired.
From this point the normal playback sequence starting with Init can proceed as long as the license acquisition was successful.
Init()
CommandCompleted(PVMFSuccess)

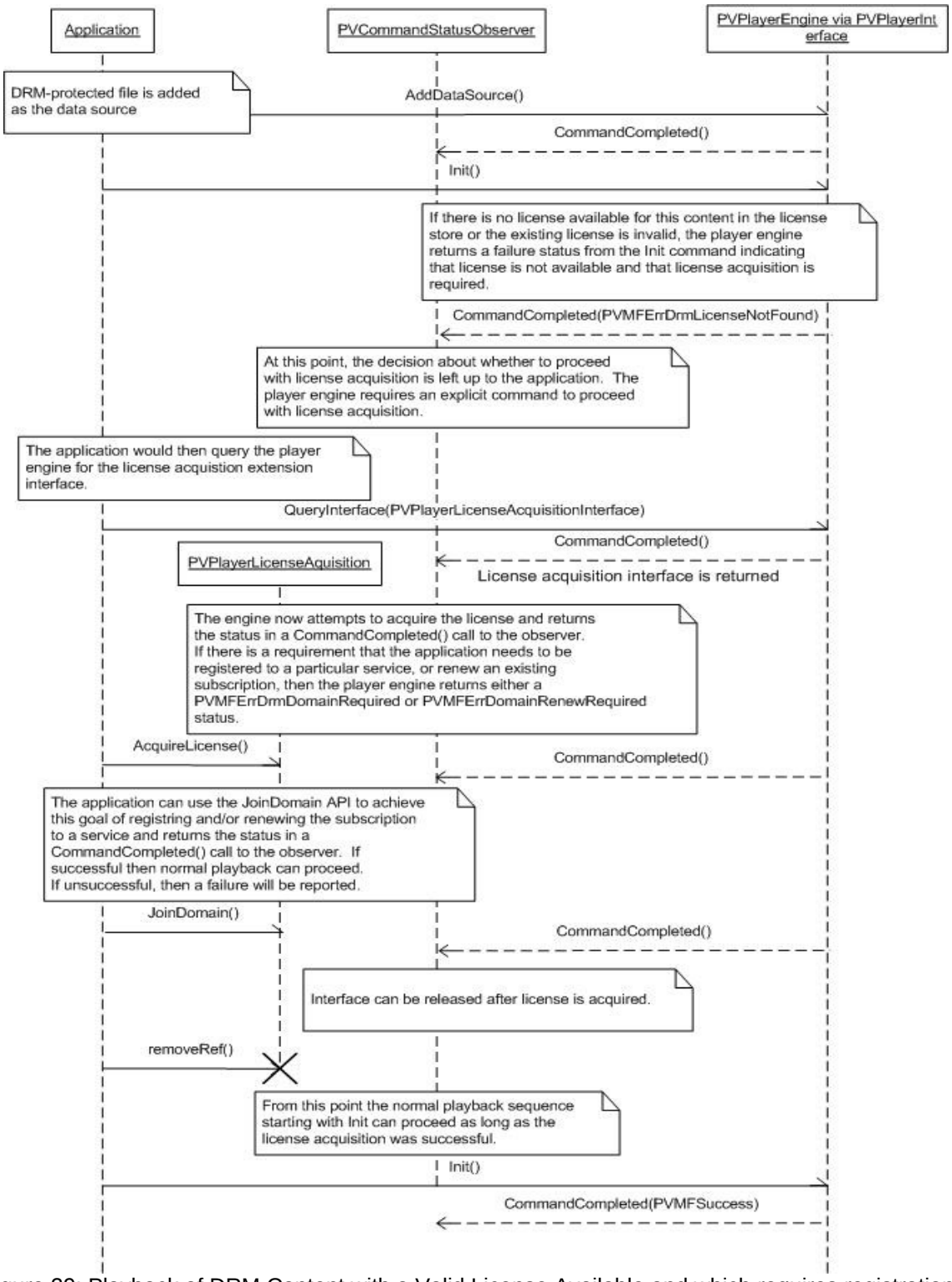



Figure 29: Playback of DRM Content with a Valid License Available and which requires registration to a service

## 13.10.4 Cancel the License Acquisition of DRM Content

The AcquireLicense call can be canceled before the engine returns the result in the CommandCompleted callback to the observer. It can make the call to CancelAcquireLicense. Once CancelAcquireLicense is called during the license acquisition, the engine attempts to cancel the license acquisition and returns the result in the CommandCompleted callback to the observer. The call to AcquireLicense will be returned with the return code PVMFErrCancelled, if cancel was successful. The sequence is shown in the diagram below.

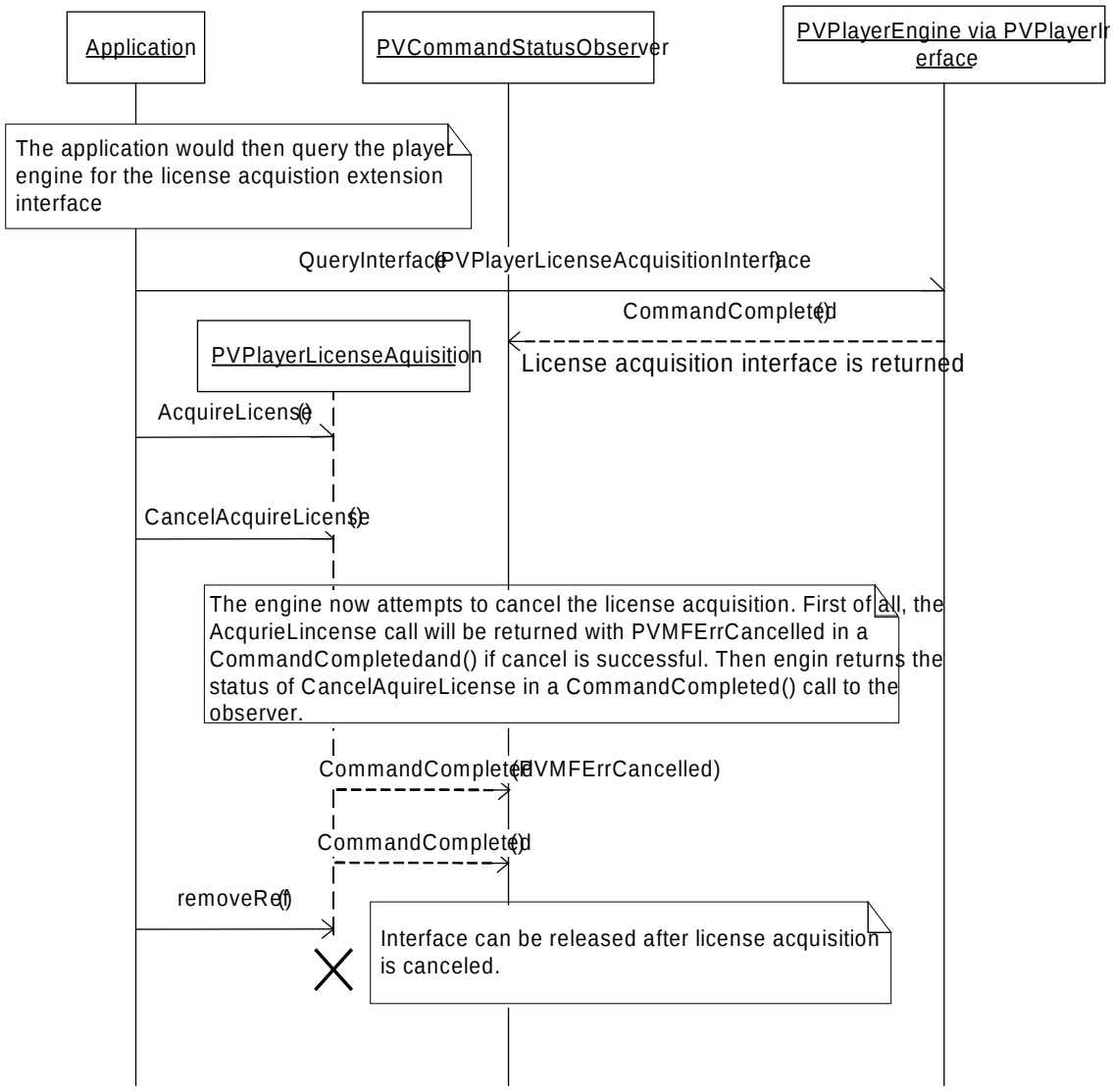



Figure 30: Cancel License Acquisition

## 13.10.5 Preview of DRM Content *without* a Valid License Available

A variation on the scenario covered in Section 13.10.3 is the case where there is no valid license for full playback of a piece of content, but there it can be previewed. This scenario might be a common way of initially distributing content so that consumers can preview it before deciding to purchase a full license. In this case the Init() method will return with the code

PVMFErrLicenseRequiredPreviewAvailable, which indicates that a license is required for full playback but a preview is available. In order to play the preview, the application must remove the current source then add it back with a flag set on the local data source to indicate preview mode. The sequence diagram below shows the interaction between the engine and the application.

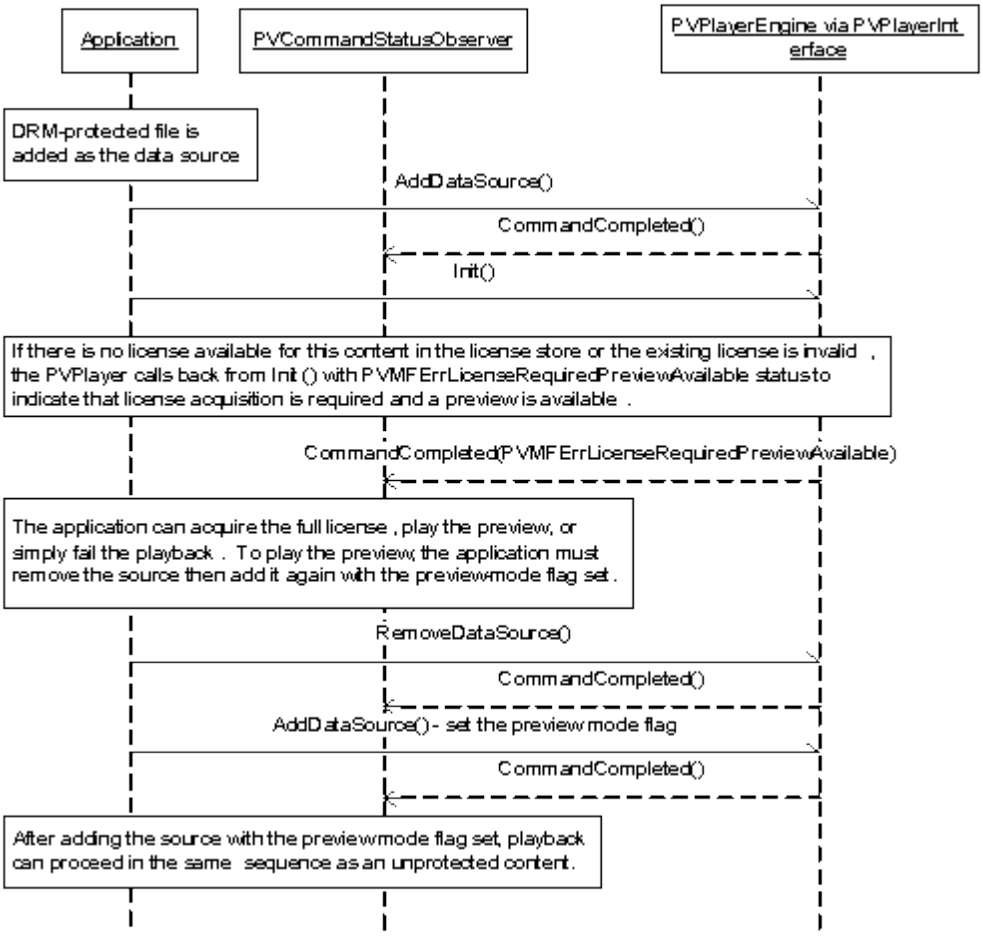



Figure 31: Preview of DRM Content without a Valid License Available

## 13.10.6 Playback of DRM Content with Auto License Acquisition

In cases where the license for the content exists in the license store, but it is invalid, the may attempt automatically acquire a valid license if it has been marked for auto-acquisition (handled outside of the player engine). In that case the engine will automatically attempt the license acquisition during the Init phase and return the status to the application in the CommandCompleted call to the observer. In the process the engine will send an informational event to the event observer as a notification that the license acquisition is happening. The sequence is shown in the diagram below.

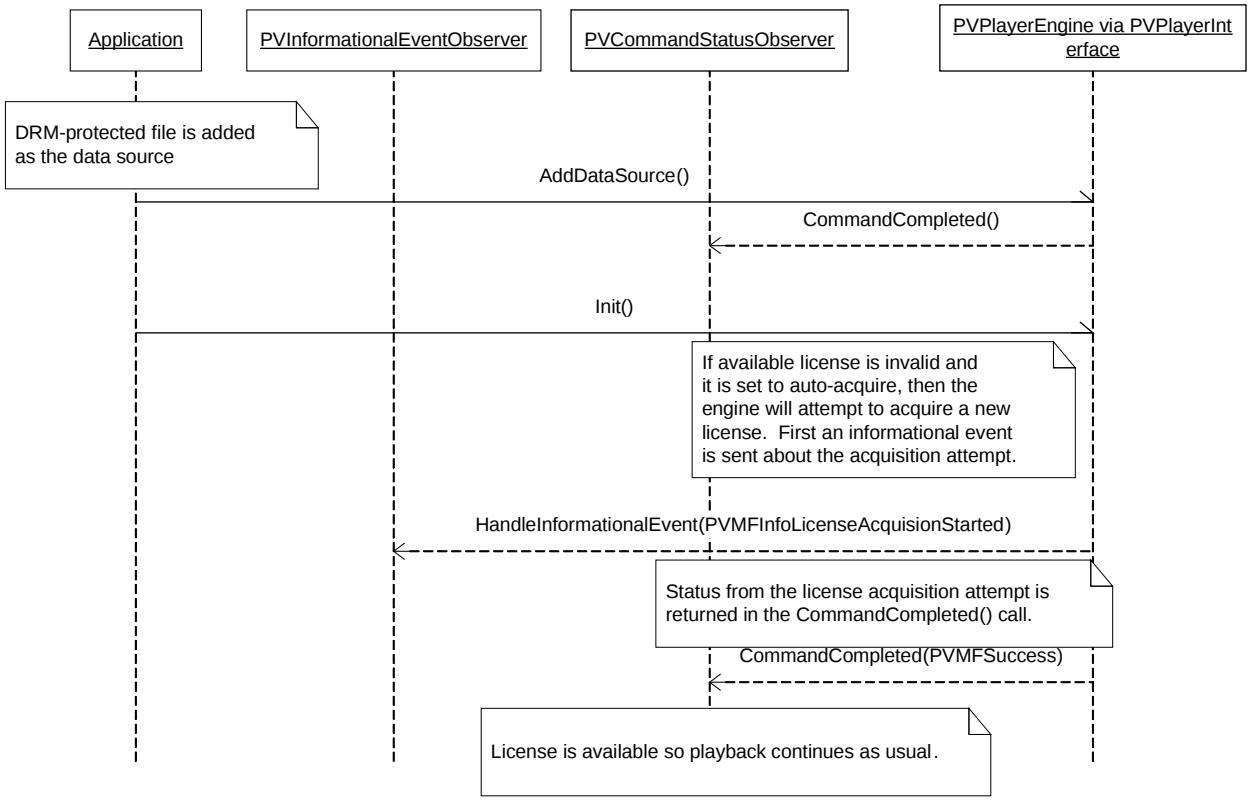



Figure 32: Playback of DRM Content with Auto-Acquisition of the License

# 13.11 Using SetPlaybackRange and PVMFInfoEndOfData Event

The SetPlaybackRange() method can be used to set the end time of playback at which point PVPlayer will send a PVMFInfoEndOfData event and stop playback. If no end time was specified, that event gets sent upon reaching the end of the stream instead. Information on how to obtain the complete duration can be found in section Metadata Handling.

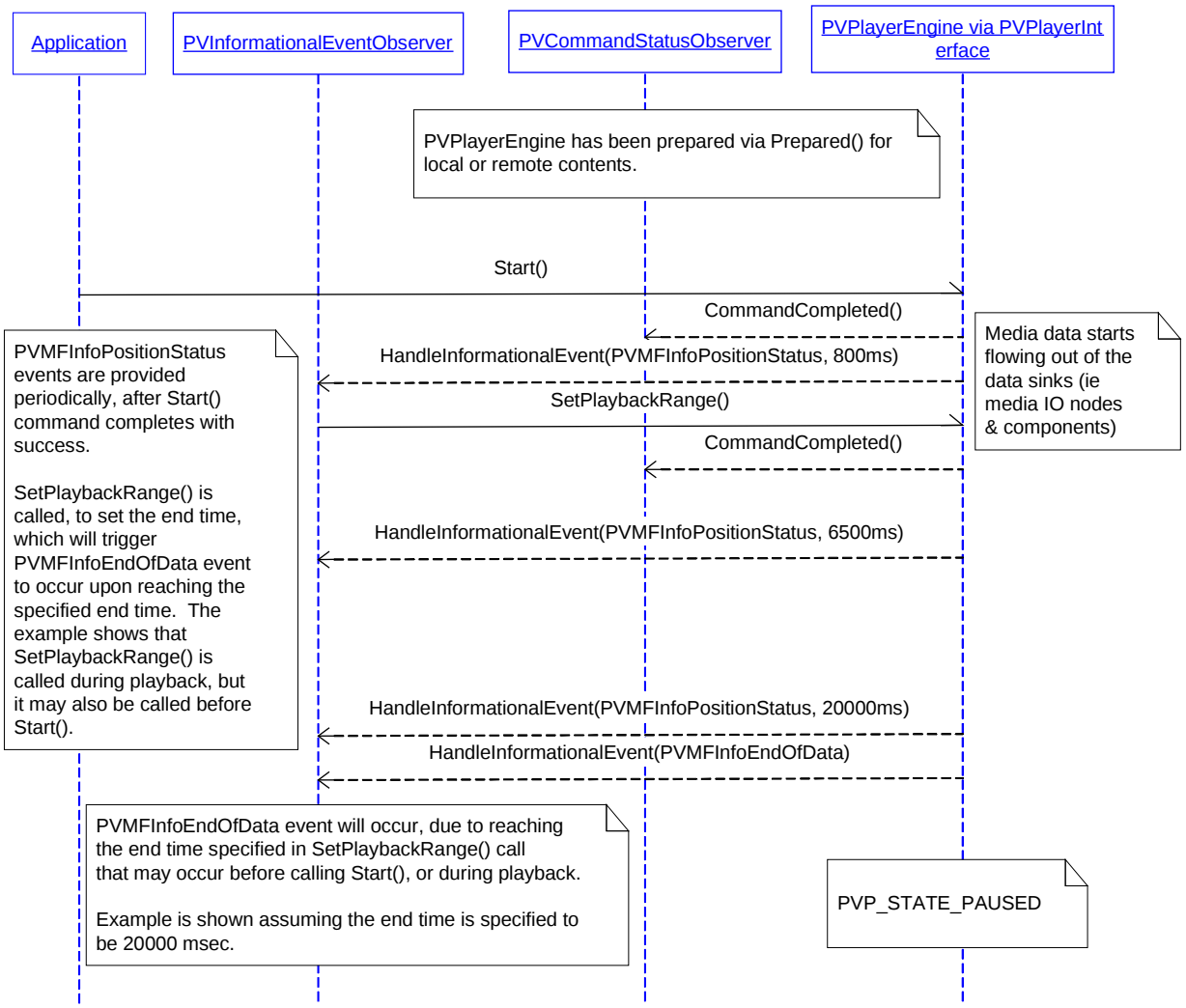



Figure 33: Using SetPlaybackRange and PVMFInfoEndOfData Event

## 13.12 Looped Playback Using SetPlaybackRange

Applications can achieve looped playback by repositioning the playback to the beginning of the file or stream using SetPlaybackRange() upon receiving the PVMFInfoEndOfData event.

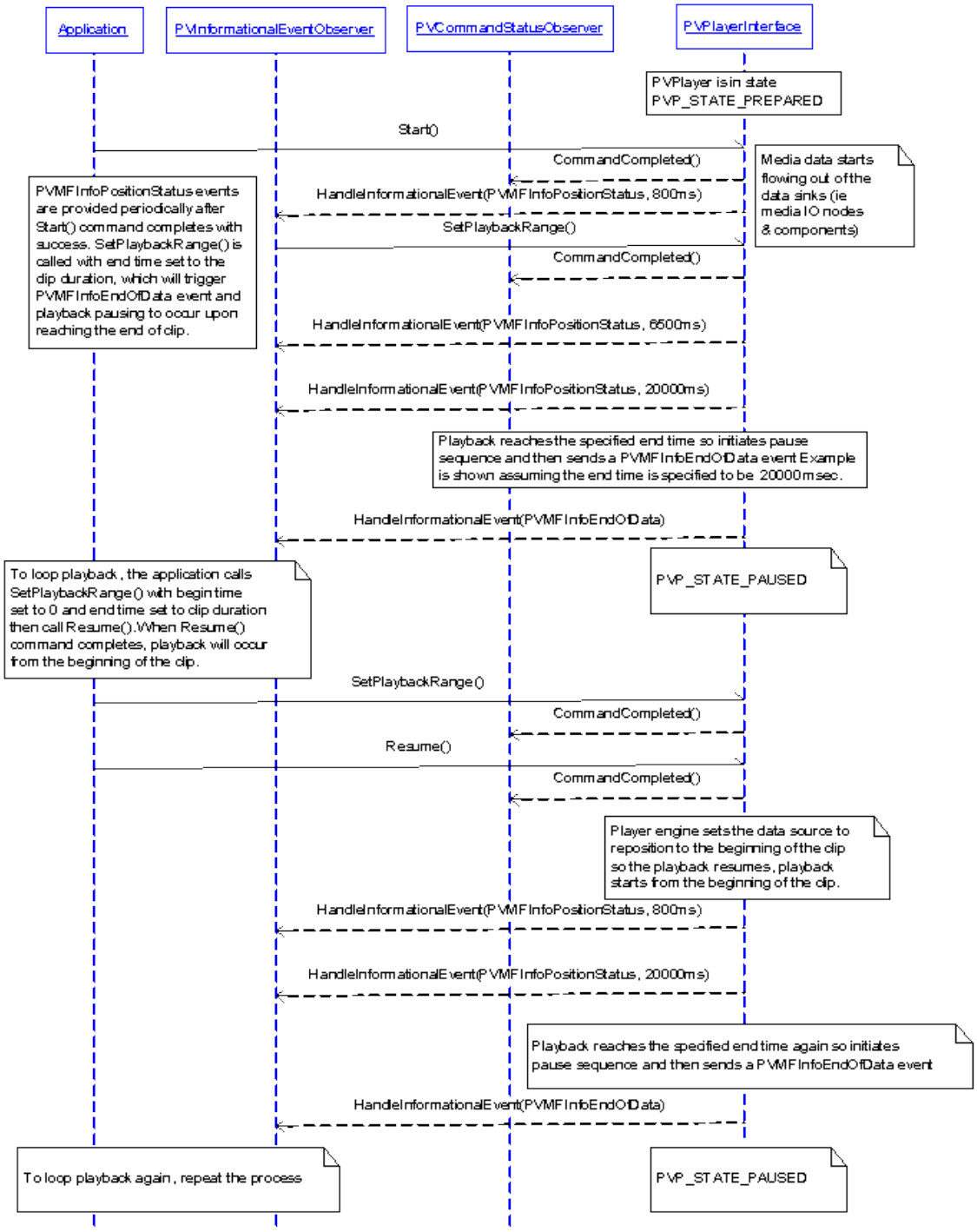



Figure 34: Looped Playback Using SetPlaybackRange

# 13.13 Start Download Session

To start a download and playback session, the application would need to configure the PVPlayerDataSourceURL object to provide the download source URL, playback control mode and other information. The following playback control modes are defined in PVMFSourceContextDataDownloadHTTP class:

1.ENoPlayback – Download only and playback will not be started
2.EAfterDownload – Playback will start after download is completed
3.EAsap – Progressive Download mode where playback starts as soon as possible.
4.ENoSaveToFile – Progressive Streaming mode where downloaded media data is not stored in file.

The sequence below illustrates the interaction between application and PVPlayer to start a download playback session.

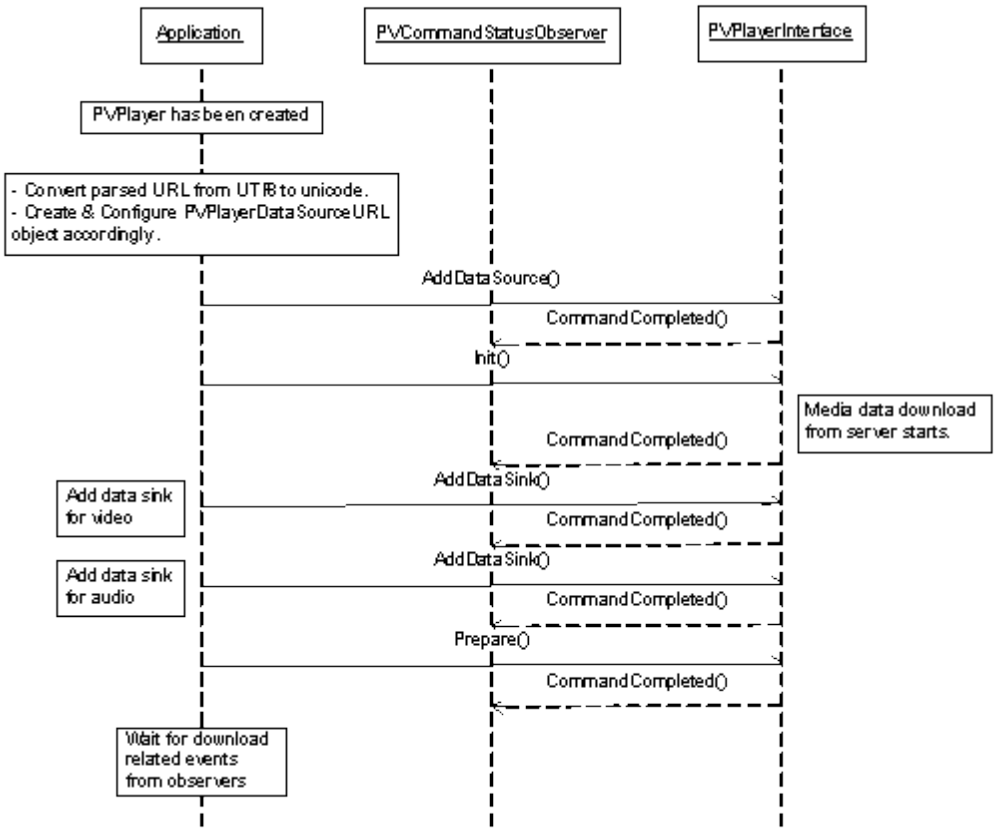



Figure 35: Start Download Session

# 13.14 Handling Download Events

This diagram shows a progressive download session that starts playback as soon as enough media data has been downloaded.

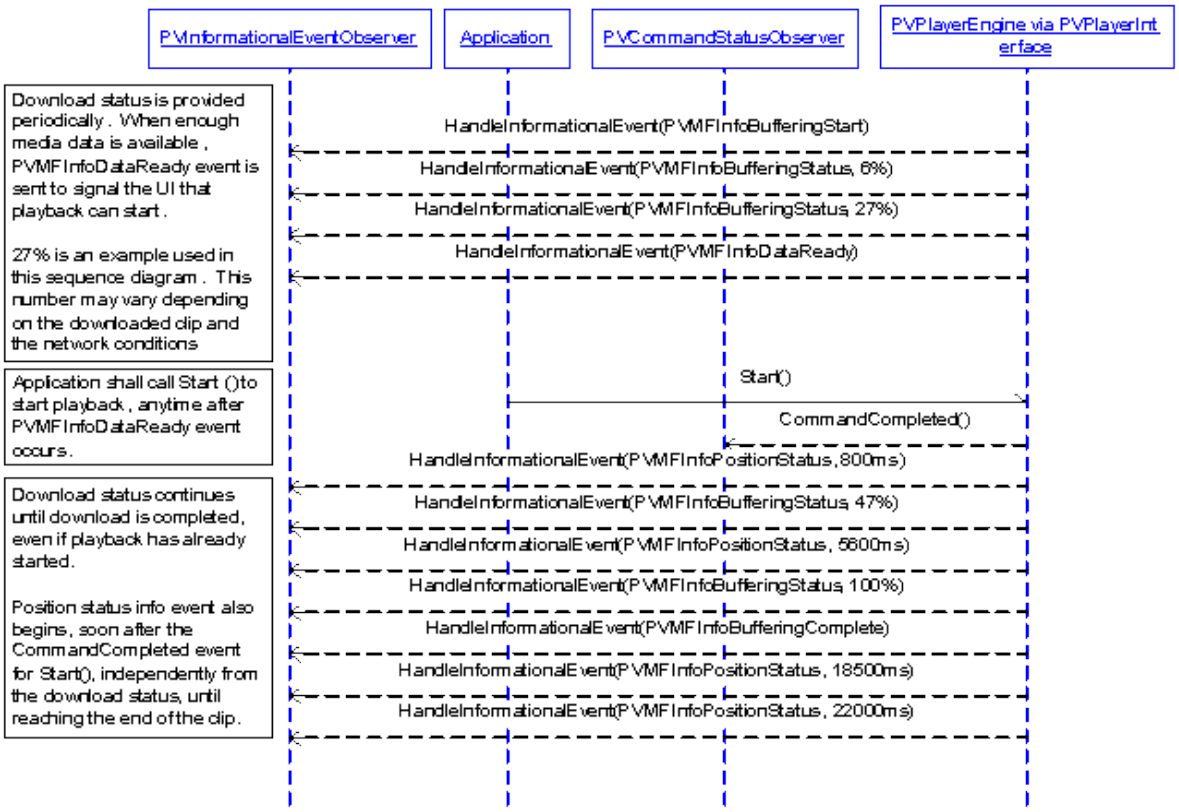



Figure 36: Handling Progressive Download Events

## 13.15 Handling Progressive Download Events

This diagram shows a download session that starts playback after the entire file is downloaded.

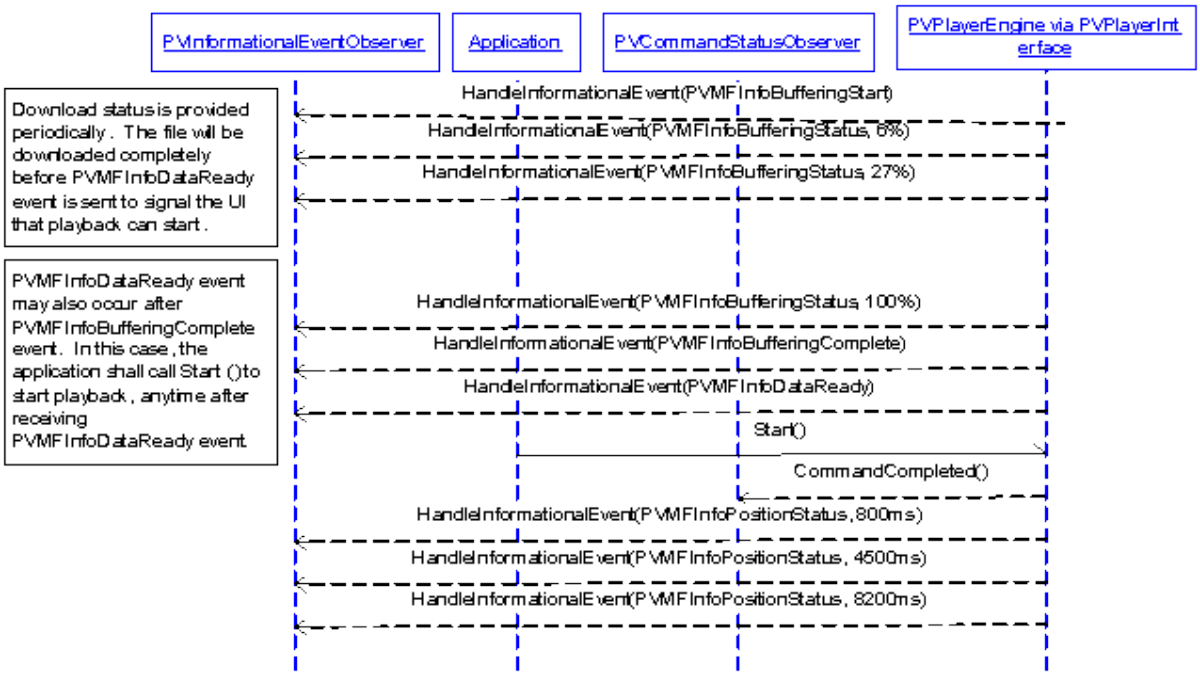



Figure 37: Handling Download Events

## 13.16 Auto-Pause-Resume in Progressive Download Session

In progressive download session, it is possible that the playback is consuming data from the partially downloaded media file faster than the download speed due to network condition and/or buffering condition settings. When PVPlayer runs out of data for playback, it would trigger the auto-pause sequence and notifies the application by sending PVMFInfoUnderflow event. When more data is downloaded during auto-pause, PVPlayer auto-resumes the playback and notifies the application by sending PVMFInfoDataReady event. It is not necessary for the application to call Start() again upon PVMFInfoDataReady event for auto-resume notification.

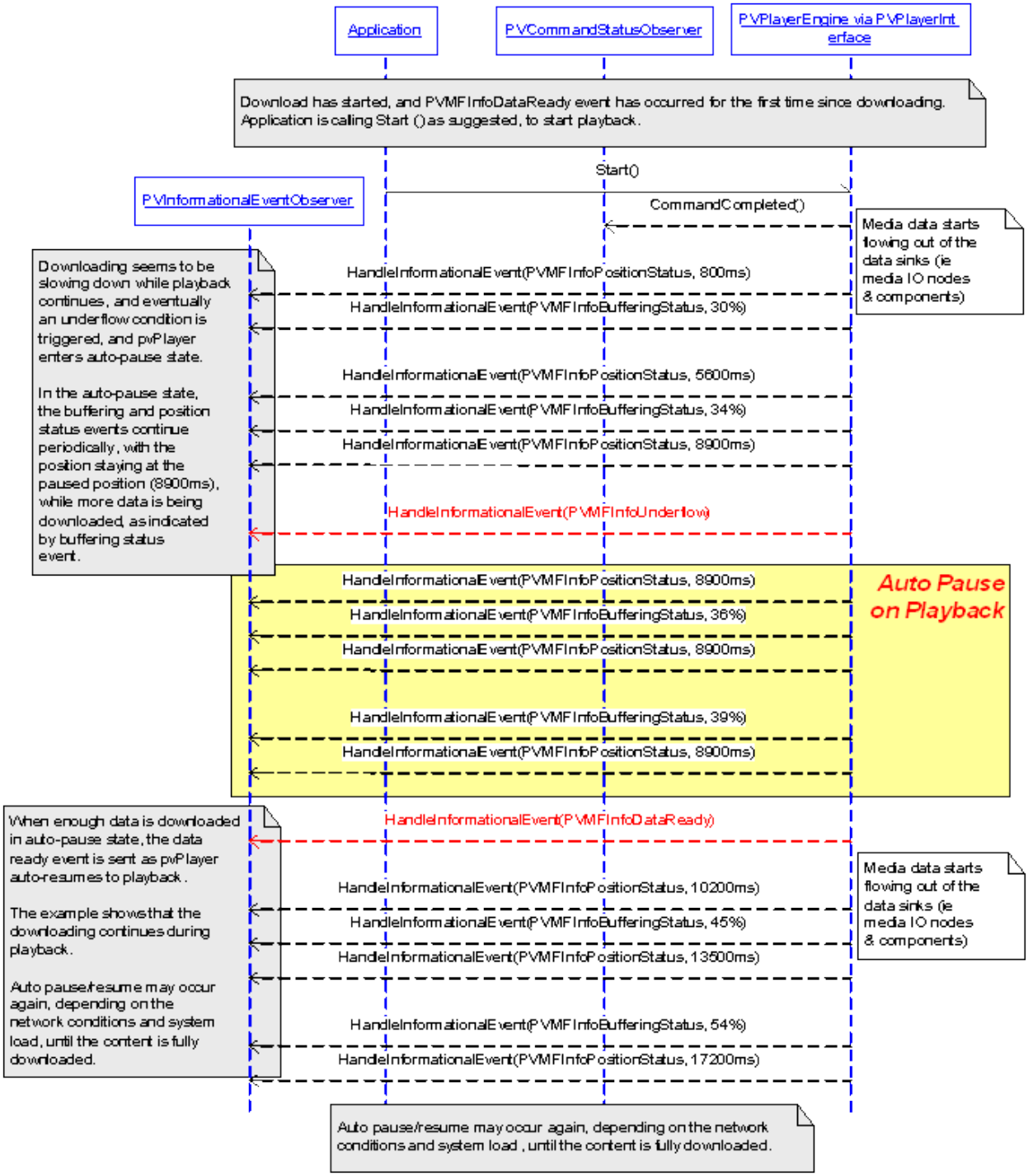



Figure 38: Auto-Pause-Resume in Progressive Download Session

# 13.17 Error Recovery During Initialization

When processing the Init() request, PVPlayer engine receives an error from the source node (e.g. file not present, network not available). PVPlayer engine goes to the ERROR state, handles the error by resetting the source node, and goes back to the IDLE state. When error handling is complete, PVPlayer engine reports PVMFInfoErrorHandlingComplete informational event.

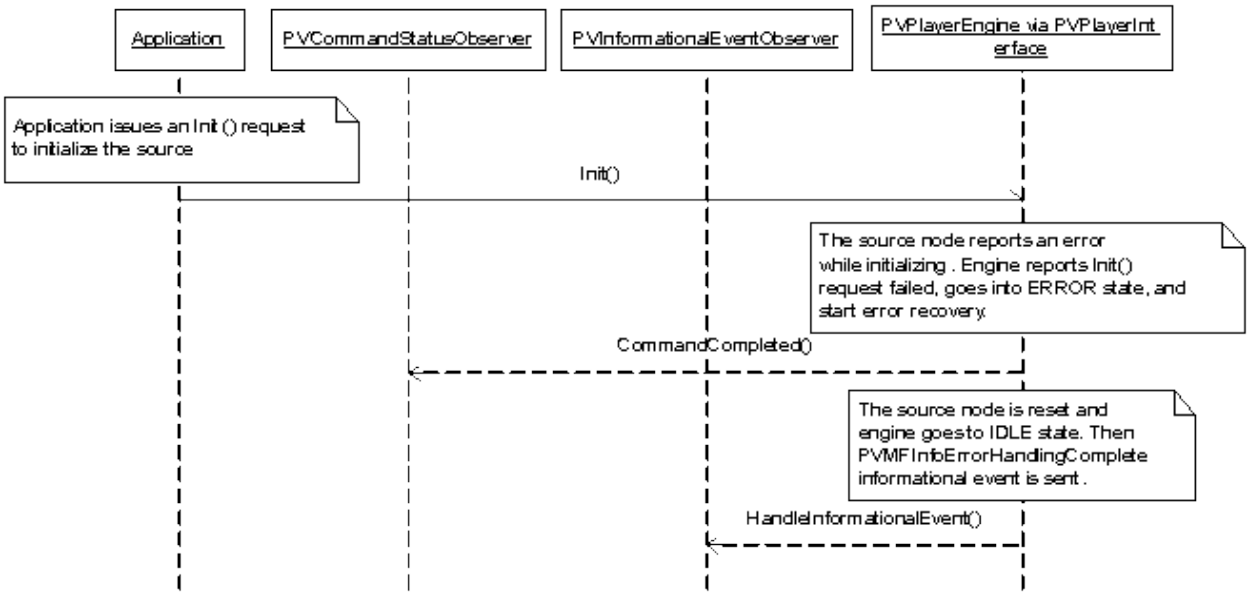



Figure 39: Error Recovery During Initialization

# 13.18 Error Recovery During Playback

During playback, PVPlayer engine receives an error from a decoder node (e.g. device became unavailable, corrupt data). PVPlayer engine goes to the ERROR state, handles the error by stopping the playback, and goes back to the INITIALIZED state. When error handling is complete, PVPlayer engine reports PVMFInfoErrorHandlingComplete informational event.

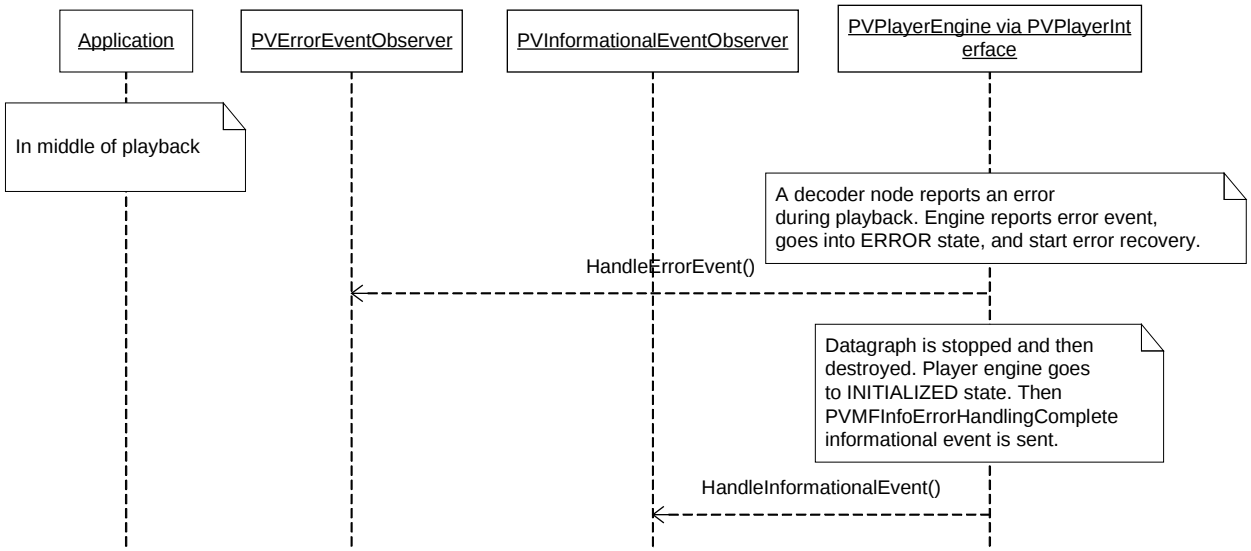



Figure 40: Error Recovery During Playback

## 13.19 Unrecoverable Error Handling

During playback, PVPlayer engine receives an error from the source node which requires the node to be destroyed. PVPlayer engine goes to the ERROR state, handles the error by stopping the playback, then resetting the source node, and cleaning up. The engine ends up in the IDLE state. When error handling is complete, PVPlayer engine reports PVMFInfoErrorHandlingComplete informational event.

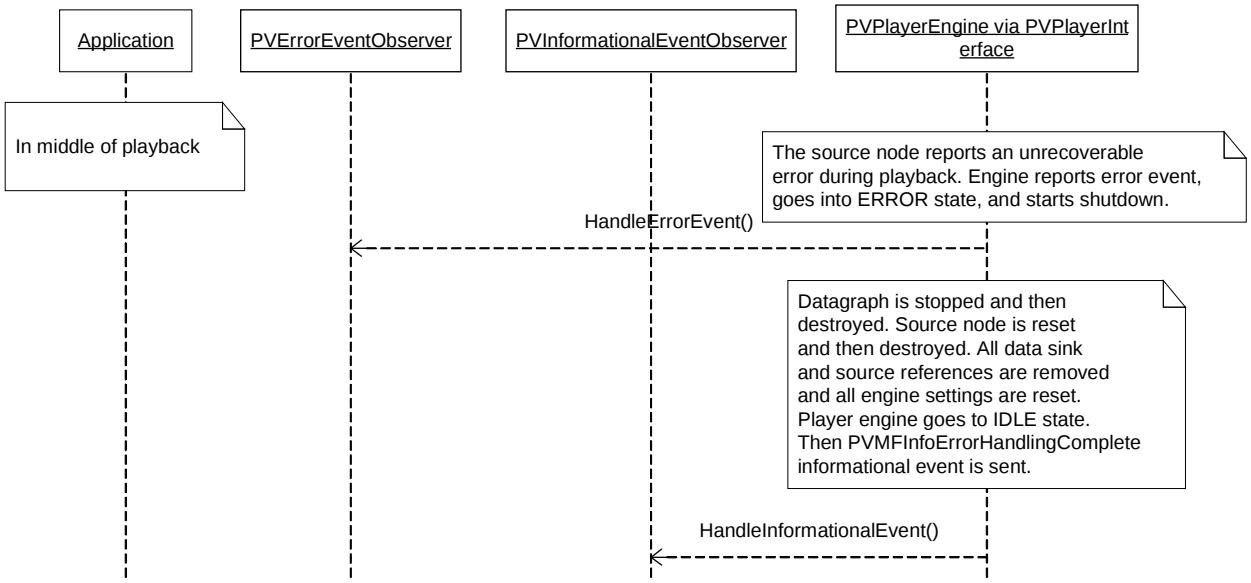



Figure 41: Unrecoverable Error Handling

# 13.20 Gapless Playback

## 13.20.1 Gapless Metadata

The PVPlayer SDK, internally, processes gapless metadata whenever available to produce true Gapless experience. The following is the information that is extracted:

Encoder Delay: Delay introduced at the start of the track, in number of samples.
Zero Padding: Padding at the end of the track, in number of samples.
Part Of Gapless Album: Whether a particular track is part of a gapless album or not.

The PVPlayer SDK currently can extract the metadata provided by either the iTunes encoder (via the ID3v2 tags) or the LAME encoder (via the XING header). Irrespective of whether a particular playback session is meant to be gapless or not, the PVPlayer SDK always removes the extra samples present at either end of the track.

Note: For cases where gapless metadata is available via both ID3v2 tags and the XING header, the preference is given to the information from the ID3v2 tags.

## 13.20.2 Gapless Playlist

Users of the PVPlayer SDK can use a single datasource to queue in multiple clips of the same format. This would constitute a Playlist session. If these clips, sequentially, are part of a gapless album, then the session can be truly “Gapless”. Using a playlist datasource eliminates delays from individual track setup & teardown that would occur if tracks are played using individual data sources. This is essential for a true gapless playback experience. ***NOTE:*** *Though there is a provision of updating the datasource after playback starts (please see Section 14.21), adding all the clips prior to the start of the playback is the preferred way of adding the datasource. Also, this is the simplest way of setting up a gapless playlist.*

The abstract class PVPlayerDataSource has additional APIs that could be used to add or append more than one clip in a datasource. These APIs are GetNumClips( ), ExtendClipList( ), SetCurrentClip( ), and GetCurrentClip( ). The following is the definition of these APIs:

```
/**
 * For playlist support
 *
 * @return
       Total number of clips in this data source.
 **/
virtual uint32 GetNumClips() = 0;

/**
 * Extends the playlist by one by adding to the end and sets the current clip to the new clip.
 * If the list cannot be expanded, the current clip is unchanged.
 *
 * @return
 *    Current clip number.
 **/
virtual uint32 ExtendClipList() = 0;
```

```
/**
 * Sets current clip number to the given index. If the index is invalid, the current clip number is
 * unchanged.
 *
 * @param aIndex
 *     Zero-based index
 * @return
 *     Current clip number.
 *
 **/
virtual uint32 SetCurrentClip(uint32 aIndex) = 0;


/**
 * Retrieves current clip number.
 *
 * @return
 *     Current clip number.
 *
 **/
virtual uint32 GetCurrentClip() = 0;
```

The following is an example of how a datasource can be extended to use it for a playlist scenario. In this example, we will be using PVPlayerDataSourceURL as a datasource which would then be used to add 3 clips in its list.

```
iDataSource = new PVPlayerDataSourceURL;
iDataSource->SetDataSourceURL(fileName1);
iDataSource->ExtendClipList();
iDataSource->SetDataSourceURL(fileName2);
iDataSource->ExtendClipList();
iDataSource->SetDataSourceURL(fileName3);
OSCL_TRY(err, iCmdId = iPlayer->AddDataSource(*iDataSource, (OsclAny*) & obj));
```

The sequence diagram is exactly the same as shown for a single local playback as shown in 72. Only difference is in the “AddDataSource()” command, where for the former we add only a single clip, and here, we can add multiple clips.

NOTE: The following are a few limitations to the playlist usage:
- Only audio clips in a playlist are supported. If a playlist contains:
    A video clip at the top of the list, only the first (video) clip will be played.
    An audio clip at the top of the list, then all non-audio only clips in the rest of the playlist will be ignored.
- All clips in the playlist should be of the same codec type.
- The codec parameters such as sampling rate, no. of channels, are also expected to be the same. For mismatched parameters, it is expected that the MIOs support the in-band reconfig message.

# 13.21 Usage of UpdateDateSource() for Playlist Sessions

In a playlist scenario, instead of adding all the clips to the datasource prior to the playback, the user of the PVPlayer SDK can also update the data source after a datasource has been added, using UpdateDataSource(), such that clips get appended in the player engine, and the playback of clips happens sequentially. UpdateDataSource() can be called at any point after AddDataSource(). If the user updates the source information of a clip currently being played, then the command fails.

## 13.21.1 UpdateDataSource() before Start() - Adding new clips

In this scenario, the user calls AddDataSource(). Then, before calling Start(), the user decides to add more clips to the datasource.

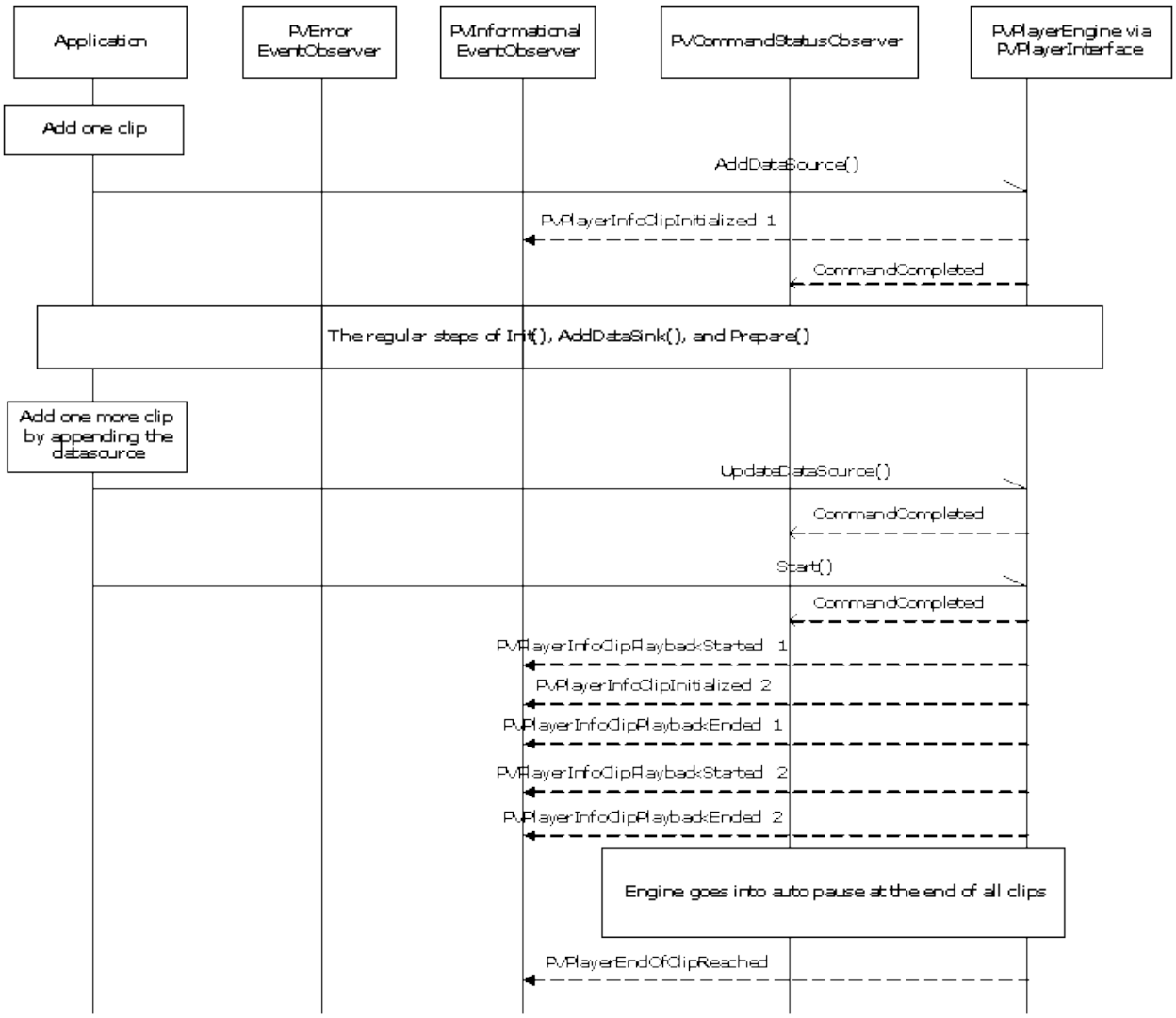



Figure 42: UpdateDataSource() before Start()

## 13.21.2 UpdateDataSource() after Start() - Adding new clips

In this scenario, the user calls AddDataSource(). Then, Start() is issued. During playback, the user wants to append clips to the existing datasource.

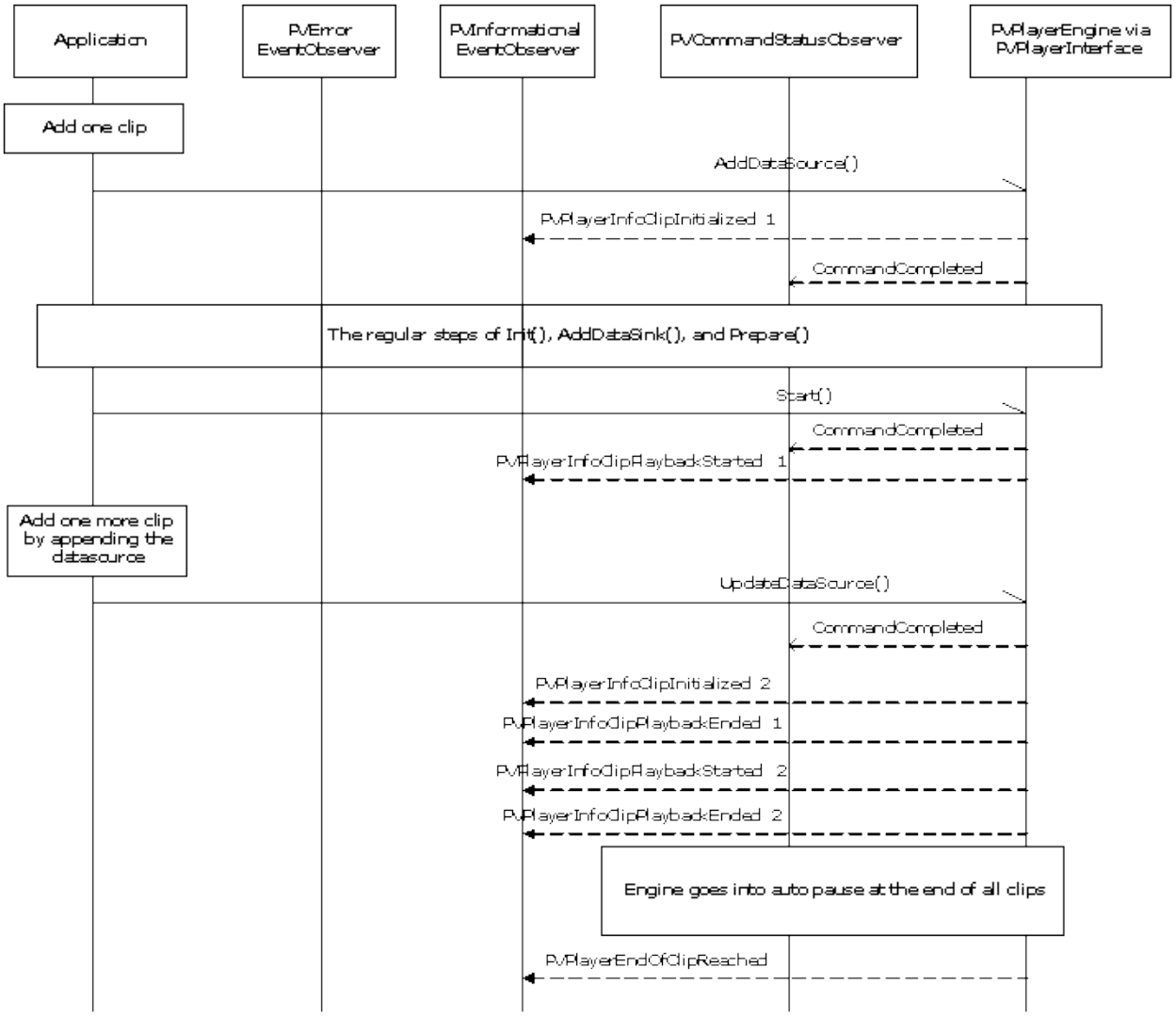



Figure 43: UpdateDataSource() after Start() - Adding new clips

### 13.21.3 UpdateDataSource() after Start() - Modifying a clip that has already started playing

In this scenario, the user calls AddDataSource() with two clips in its datasource. Then, Start() is issued. First clip finishes playback, and the second clip starts. At this point, the user calls UpdateDataSource() with the clip in slot 2 modified. It is expected that UpdateDataSource() in this scenario fails.

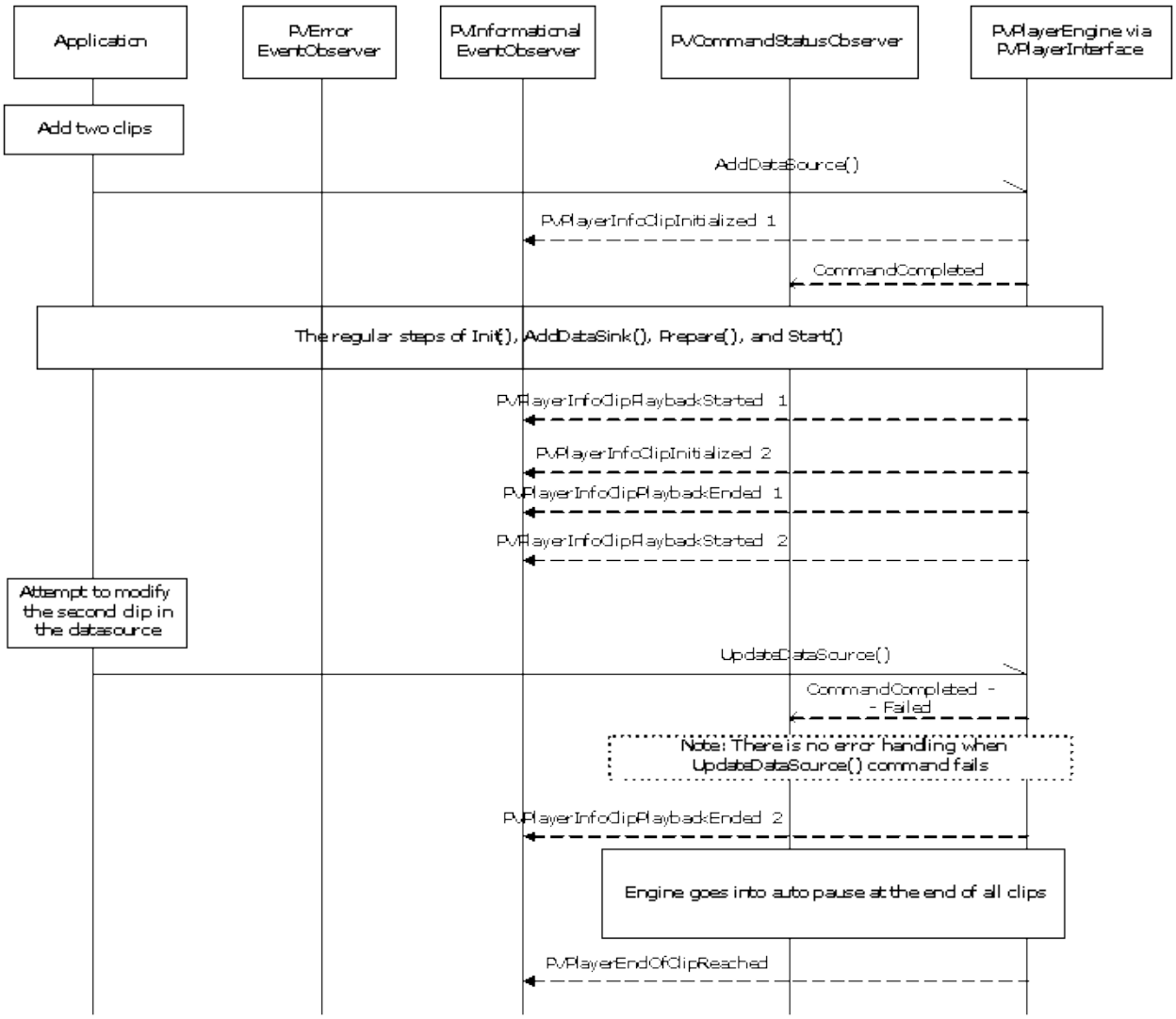



Figure 44: UpdateDataSource() after Start() - Attempt to modify a clip that has already started playing

# 14 Applications Involvement in Track Selection

It is possible for the user of PVPlayerSDK (also referred to as application or app) to participate in the track selection process. The extension interface (PVPlayerTrackSelectionInterface) is exposed via the player API, QueryInterface(), by requesting with the UUID associated with the interface. Before describing this interface and its usage we define the following terms:

- Complete List – This is the complete list of available tracks in multimedia presentation.
- Playable List – This is the list of playable tracks. The list of playable tracks is a subset (at times a proper subset) of the complete list of available tracks.
- Selected List – This refers to the list of tracks selected by player engine from playable list for playback. Player engine performs track selection during “prepare”.
- PVMFMediaPresentationInfo – Player engine uses this data structure to expose the contents of the above three lists.
- PVMFTrackSelectionHelper – This is an abstract interface that exposes a synchronous method to obtain track selection inputs from the user of PVPlayerSDK. If the user of PVPlayerSDK wishes to participate in the track selection process then, the user of the SDK needs to provide an implementation of this object. If provided, PVPlayerSDK will invoke the SelectTracks(...) API as part of “prepare”.

PVPlayerTrackSelectionInterface exposes APIs to retrieve the complete list, playable list and selected list. In addition to these APIs this interface also provides a mechanism for the application to register its implementation of the PVMFTrackSelectionHelper interface. While invoking the “SelectTracks(…)” API, PVPlayerSDK provides it with a playable list and the implementation of PVMFTrackSelectionHelper is responsible for creating the selected list based on this input. For exact syntax of these interfaces and their APIs please refer to \engines\player\include\pv_player_track_selection_interface.h

## 14.1 Memory Considerations

All of the APIs of PVPlayerTrackSelectionInterface and PVMFTrackSelectionHelper objects allocate memory. Therefore both these interfaces contain explicit release APIs. These release APIs remove any ambiguity about memory ownership.

# 15 Diagnostics

## 15.1 Instrumentation and Debug Logs

The PVLogger component is used inside PVPlayer SDK to log stack trace, warnings, error, and other information. PVLogger provides a very flexible and extensible framework that allows fine-grained control of the exact logging point and logging level along with the ability to filter messages and route messages to arbitrary outputs.

OSCL-based PVPlayer engine expects PVLogger to be initialized by the user of PVPlayer SDK and it provides standard SDK APIs for logging via the extension interface. Refer to the PVLogger User's Guide for more details on PVLogger.