Exhibit 39

PVPlayer SDK Developer's Guide
OHA 1.0, rev. 1
Oct 20, 2008

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.

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 platforms and 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 and data sources (e.g. multimedia file, streaming server) and 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 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) and the PacketVideo Multimedia Framework (PVMF). 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.

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.

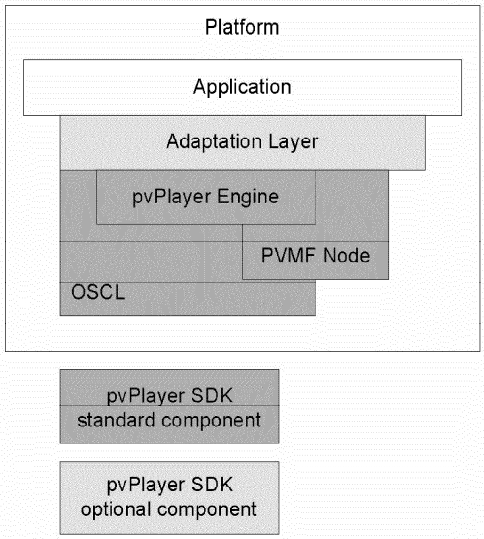

Figure 1: PVPlayer SDK Software Stack

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.

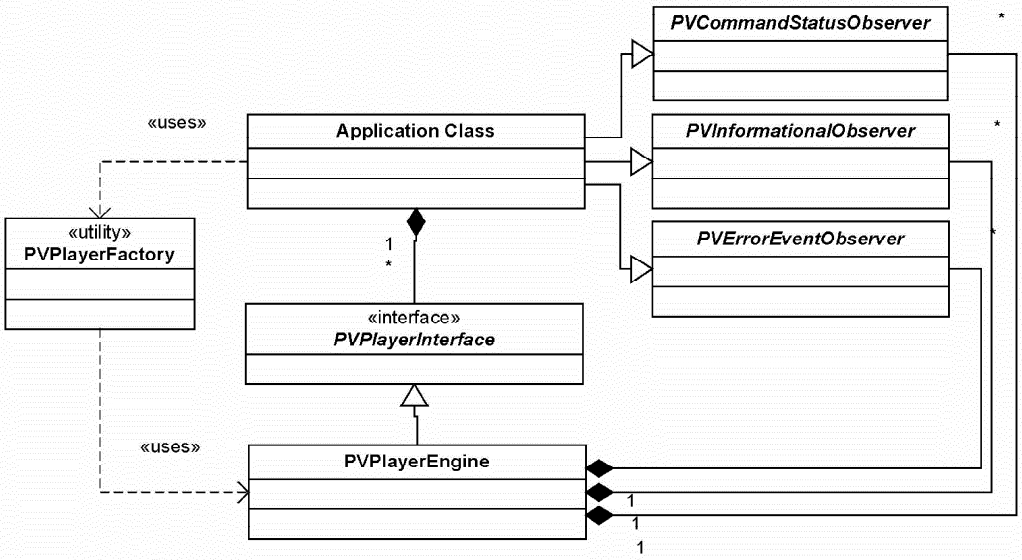

Figure 2: Class Diagram

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.

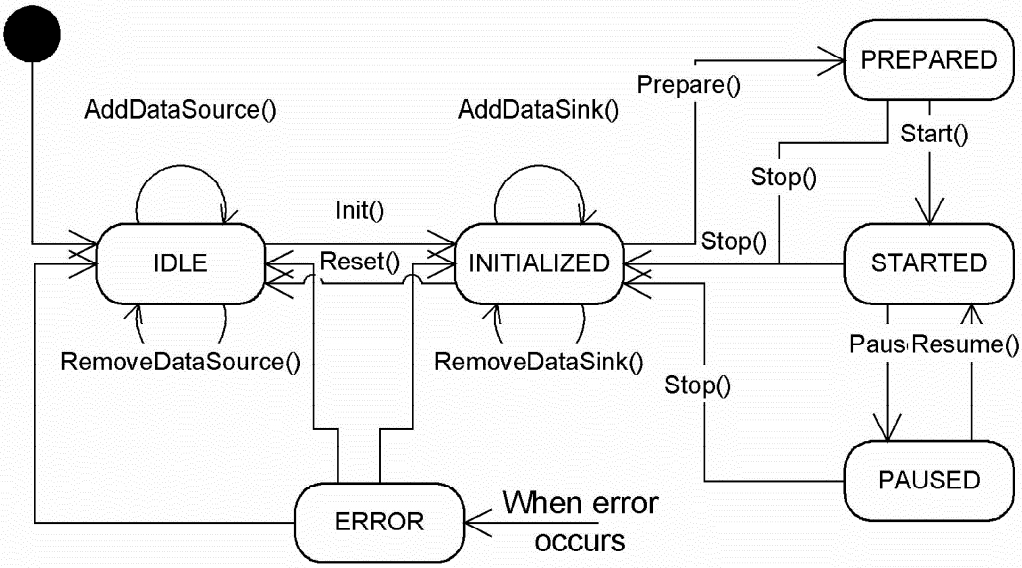

Figure 3: State Transition Diagram

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.

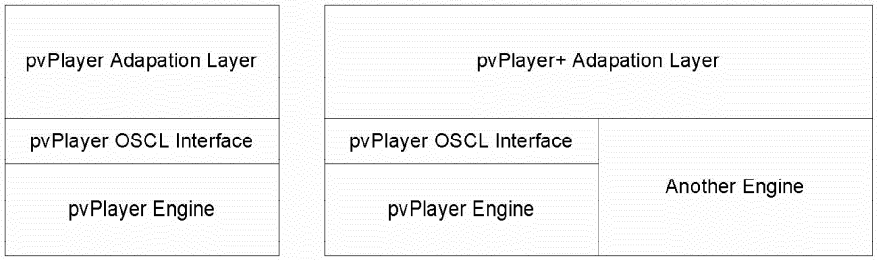

Figure 4: PVPlayer Adaptation Layer

4.3 Multi-Threading Support

The default OSCL-based interface is not multi-thread-safe. To have multi-threading support in the interface, the adaptation layer would need to provide such a feature. One method is to use OSCL proxy interface component to provide multi-threading support. Other method is to add platform specific multi-threading support for a particular platform to PVPlayer SDK's adaptation layer. The diagram below shows how multi-threading support would be accomplished via the two methods. In the left block diagram, the adaptation layer utilizes the OSCL proxy framework, which minimizes platform specific coding in the adaptation layer by pushing platform specific code to OSCL. In the right block diagram, the adaptation layer directly uses the platform threading functionality so the adaptation layer becomes platform specific.

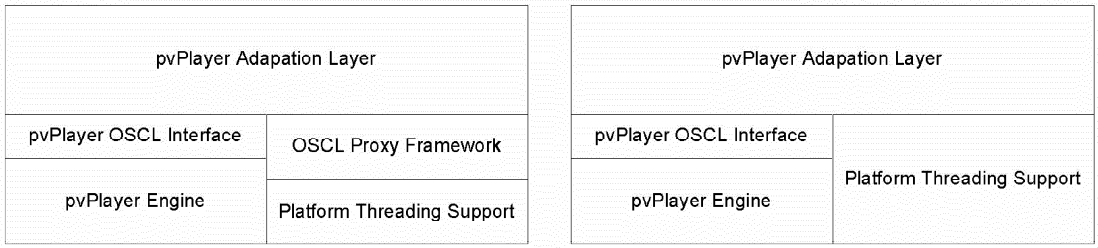

Figure 5: Multi-Threading Support

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.

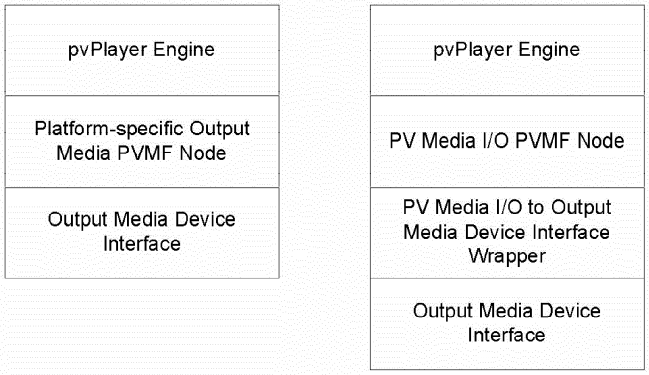

Figure 6: Media Output to Node and Media IO

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 PVMF Nodes for Player

This section gives a brief description of PVMF nodes used by PVPlayer engine. Only PVMF nodes based on OSCL and PVMF components are covered. No platform specific PVMF node is covered. For more detailed information on a particular node (one below or platform specific one), please refer to the documentation for that node.

5.1 Data Sink Nodes

Data sink node are the end points of the data-flow graph and takes the media data out of PVPlayer engine.

5.1.1 PVMFMediaOutputNode

PVMFMediaOutputNode is a wrapper node around the PV media I/O interface to output data. The node translates node commands and incoming media data to appropriate media I/O actions and handles media I/O events. Using PVMFMediaOutputNode allows encapsulation of platform and device specific output interface with PV media I/O interface.

5.1.2 PVMFFileOutputNode

PVMFFileOutputNode accesses the file directly using OSCL file I/O to write media data coming in via the port. The node has some capability to understand format type and to write out data appropriately for the specified format type (AMR file header for AMR IETF format).

6 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 this section.

6.1 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 OSCL clock which provides a control to set, start, pause, stop, and adjust the clock. OSCL clock also allows the timebase to use for the clock source to be specified by PVPlayer engine. For more information OSCL clock, please refer to its design document.
PvPlayer engine creates an instance of OSCL 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.