PVMFMediaClock Guide

OpenCORE 2.0, ver 1

Feb 3, 2009

Table of Contents

List of Figures

1. Introduction

This document describes the details of the updated media clock, PVMFMediaClock, introduced as part of the OpenCORE 2.0 release. The media clock is responsible for maintaining a common time reference used to pace media playback or capture and implement A/V synchronization. The details of the functionality and design of the PVMFMediaClock are described in the later sections of the document after briefly giving some history on the previous clock implementation to motivate the reason for the change.

The previous implementation of the media or playback clock functionality in OpenCORE 1.0 was partially contained in the OsclClock class in the oscl util layer and partially implemented in various nodes and components in the graph and within the player engine. The OsclClock was a passive library that did not directly handle active timing notification. Instead, it provided back the current media clock value when queried by an outside caller. A component waiting on the media clock to reach a particular value had to essentially poll the OsclClock component to determine if a particular value had been reached. However, the OsclClock did support observer class registration for notification of state changes. The state change notifications happened as a pass-through within the state change method, so the clock component did not need to be active.

However, the OsclClock implementation resulted in a number of issues that could be improved:

- Each component wanting to base activity on specific values (or ranges of values) of the media clock must maintain its own timers which are based on the system clock. The result is that whenever the media clock value is updated, those components must be notified and they have to cancel the current timers and most likely create new ones.
- The logic for timestamp and clock comparisons was duplicated in multiple places even though this is a very fundamental functionality. Handling that in a common location helps minimize the chance of mistakes and makes the code more readable.
- There was no central place to account for fixed, known rendering latencies in the media sinks. If all rendering components would register these values with the common clock, then the relative differences could be accounted for when dealing with requests for clock value notifications associated with rendering decisions.
- The mapping between the media clock value and the Normal Play Time (NPT) or clip position was handled outside the clock component. It was best to handle this within the clock since it made notifications based on NPT possible.

The new PVMFMediaClock is enhanced to actively manage notifications based on clock values. It maintains a single timer representing the earliest deadline among all pending requests. The code for the updated media clock is located in the PVMI layer rather than OSCL (where the OsclClock was located) since the logic is not part of the OS abstraction and fits best as part of the multimedia infrastructure layer.

2. PVMFMediaClock Features

This section describes the features of PVMFMediaClock.

2.1. Timekeeping

The PVMFMediaClock acts as a time source for multimedia graphs. There is a provision for specifying a timebase which the clock uses as a time source. This timebase can be a system clock or any other time source (e.g., audio driver clock). The media clock itself is derived from PVMFTimebase; therefore one instance of PVMFMediaClock can act as a timebase for another.

Besides APIs to get the current time, the media clock also provides APIs to start, stop, adjust, pause, etc the clock. A user can specify time values in various units with microseconds being the smallest unit of time.

2.2. Clock Observers

The PVMFMediaClock can notify objects about changes in the clock state. The objects can set themselves as clock observers to receive notifications by implementing observer interfaces and calling the API to set observer. There are three observer interfaces.

1.PVMFMediaClockObserver - notification for clock timebase update, clock count update, clock adjustment.

2.PVMFMediaClockStateObserver – notification for clock state changes.

3.PVMFMediaClockNotificationsObs – For getting callback notifications.

Objects can implement more than one of the observer interfaces to get multiple notifications.

2.3. NPT Mapping

The media clock implements a monotonic non-decreasing counter that represents a time reference to be used for media rendering decisions. It does not directly provide any reference to the clip position (a.k.a., the normal play time (NPT)). If there is random positioning in the clip, then the NPT may jump ahead discontinuously or even jump backwards, but the media clock would continue to move ahead in a continuous manner. Therefore a mapping must be maintained to relate a given media clock value back to the corresponding NPT value. Figure 1 below illustrates how the mapping between the times works.

The PVMFMediaClock maintains a mapping between the playback time and the NPT. Any module changing the clip position must notify the media clock about clip repositioning events using the UpdateNPTClockPosition() API so that the NPT mapping can be maintained.

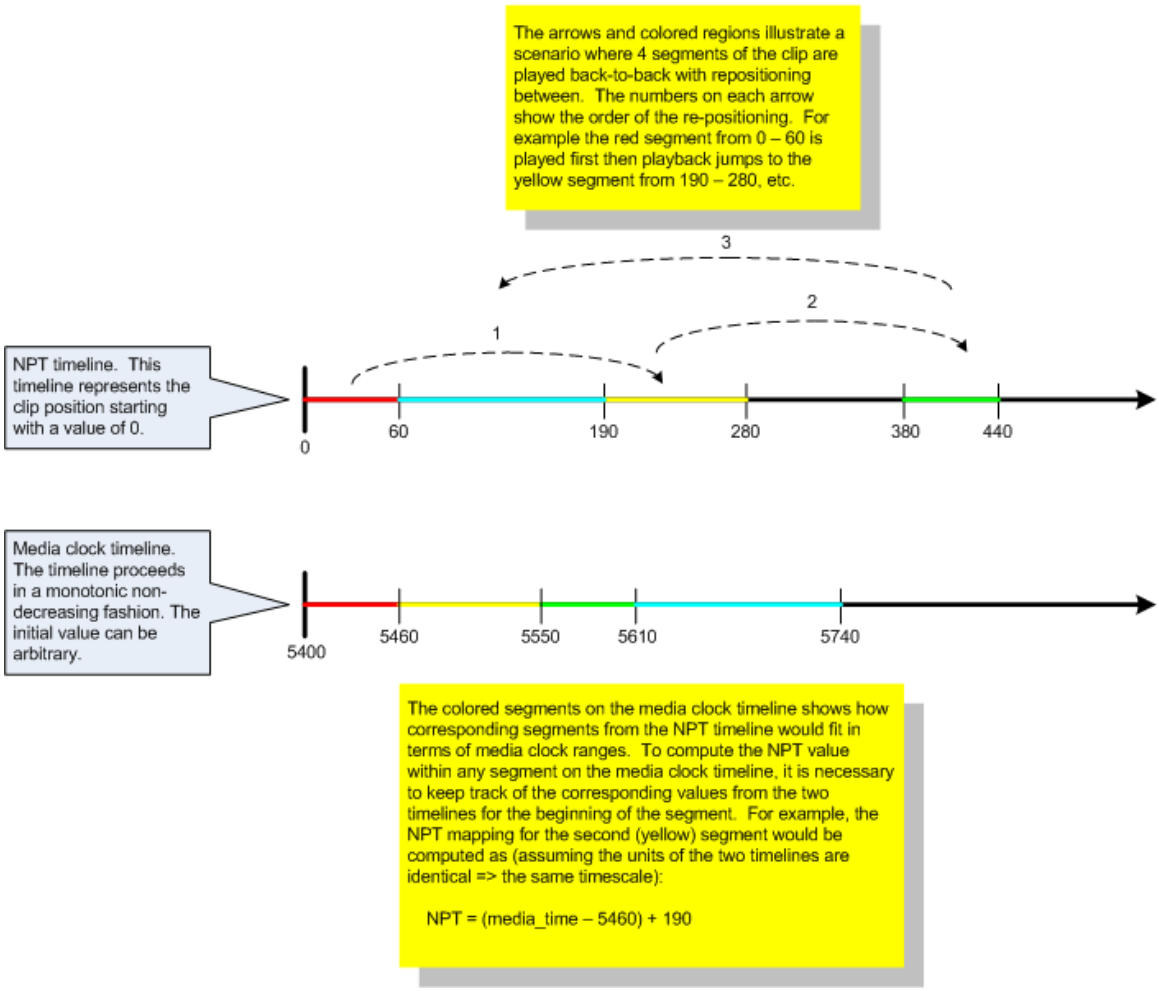

Figure 1: Illustration of the Mapping between the Media Clock and NPT.

2.4. Timer Callbacks

Components that need to take action based on the value of the media clock can set callbacks on PVMFMediaClock. These callbacks eliminate the need for components to set their own timers and cancel and reset them to a new value when clock value is adjusted or clock is paused. Callbacks can be set to an absolute media clock time or a delta from the current media clock time when the timer should call back. A callback timer can be canceled after it has been created. It is also possible to set/cancel callbacks based on NPT clock time.

Instead of an absolute time, the media clock takes an input specifying a time window for scheduling a callback. This approach is used since competing tasks and threads may delay a timer callback from happening at the exact requested moment, so firing a bit early may be preferable to always firing late. Also the media clock can combine firing callbacks that have overlapping windows, which reduces overhead.

2.5. Latency Handling

One common issue when integrating different sink implementations is that each may have some amount of measurable latency from the time a decision is made to render a sample to the time it reaches the final output (e.g., display, speakers, etc). These latencies need to be compensated in order for the different media types to rendered in a synchronized manner. For example, consider a case where there is an audio, video, and text sink which must be rendered in a synchronized fashion. The fixed latencies associated with these media sinks is 10, 20, and 2 msec respectively. The means that video needs the biggest head start relative to the other two in order to display a sample at the same time as the others. If all three have initial samples with timestamp values of 0, then in order to render simultaneously, video should start 10 msec before audio and 18 msec before text. The scenario is shown in Figure 2.

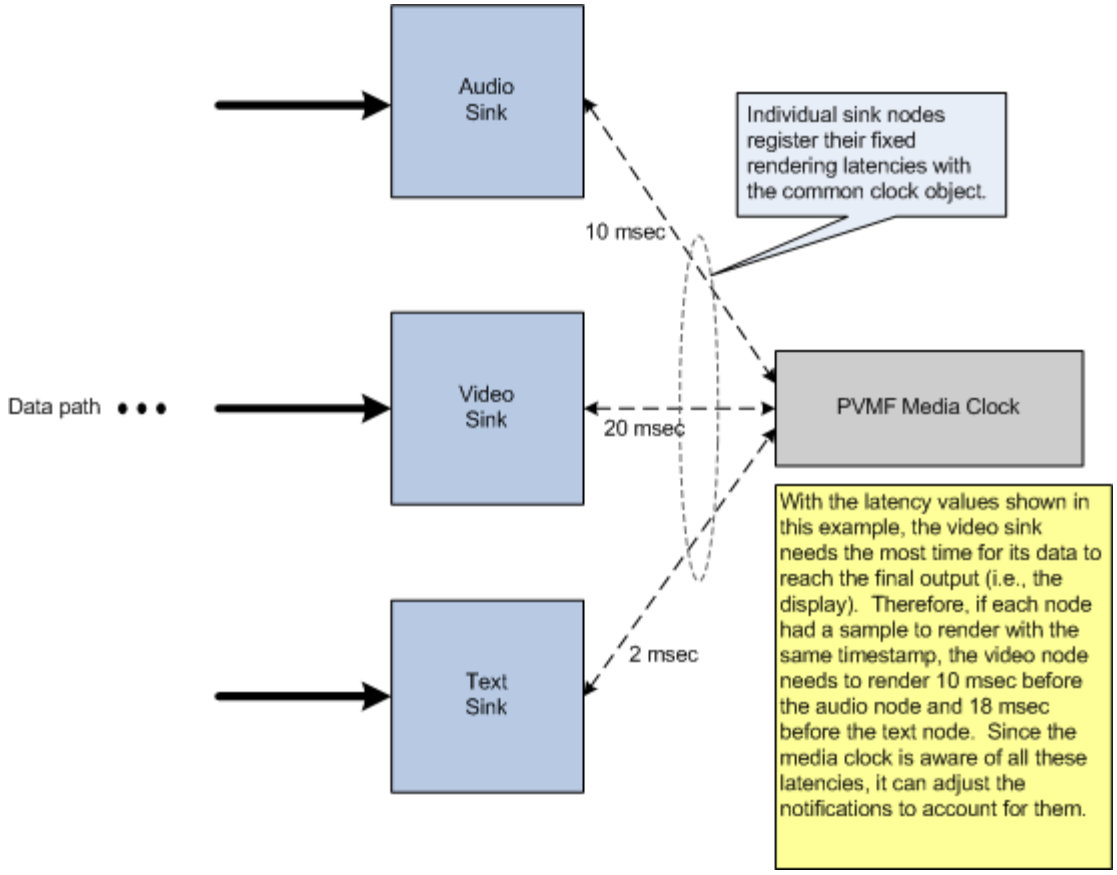

Figure 2: Sink Latency Registration

Each sink registers its latency with media clock while creating the notifications interface. The media clock subtracts the largest common latency out and adjusts the residual latency value in scheduling notifications.

2.6. NPT Clock Transition

The PVMFMediaClock has a provision to schedule an NPT clock change event in the future. A user can specify an absolute media clock time value when the change should take place along with the new startNPT value that should be used. Also, the user can make the NPT clock run in either direction, forward or backward, from that point onwards.

3. Design Details

3.1. PVMFMediaClock Class Design

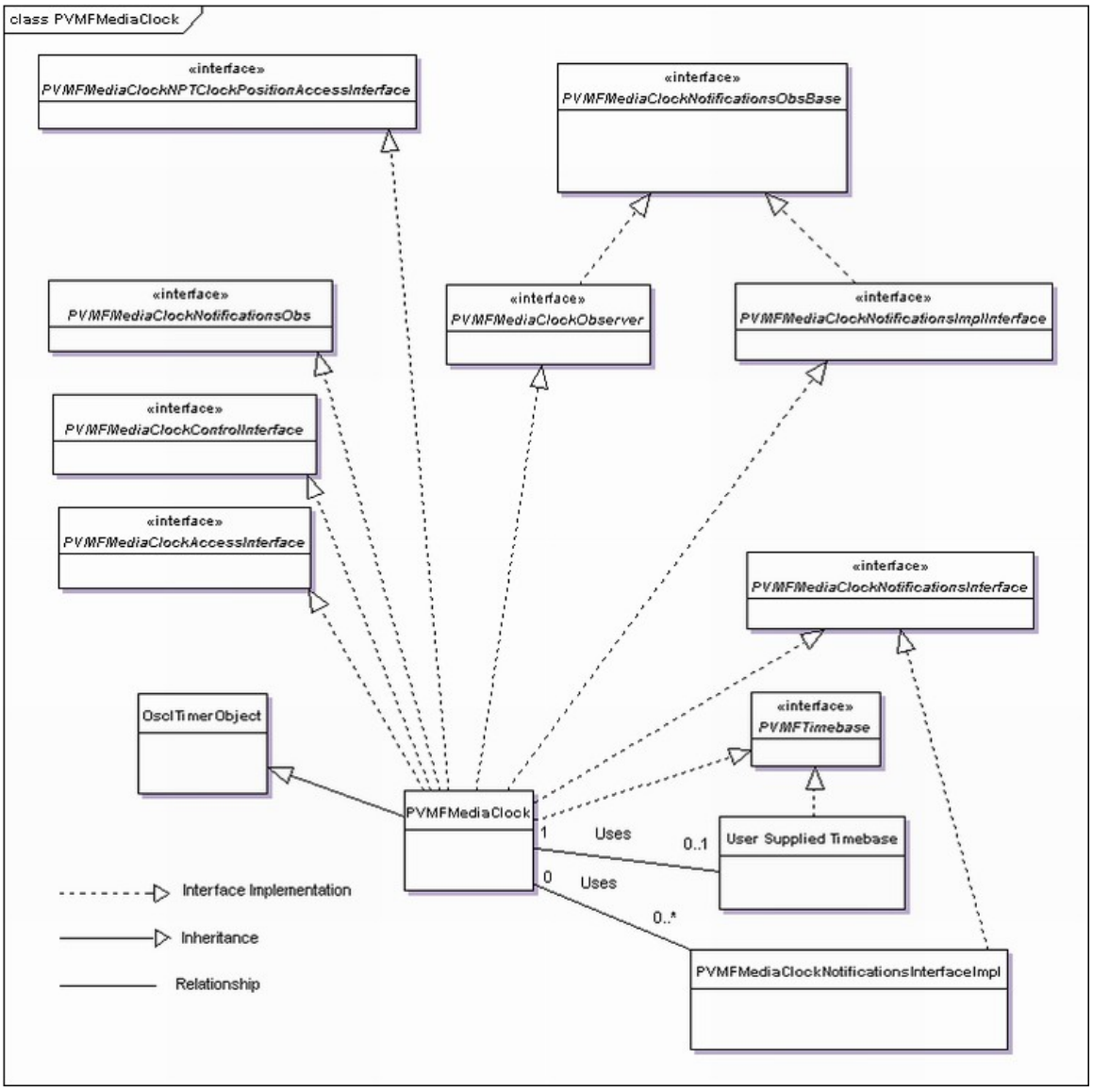

Figure 3: PVMFMediaClock Class Diagram

Figure 3 shows the detailed class diagram of PVMFMediaClock. In the diagram, dotted lines represent interface implementation, solid lines with block arrow represent inheritance and simple solid lines represent relationship between classes.

PVMFMediaClock's features are grouped into four different interfaces.

1.PVMFMediaClockControlInterface – Clock control function like Start/Stop etc
2.PVMFMediaClockAccessInterface – Time access functions like GetCurrentTime32
3.PVMFMediaClockNPTClockPositionAccessInterface – NPT mapping related APIs
4.PVMFMediaClockNotificationsInterface/PVMFMediaClockNotificationsImplInterface – The PVMFMediaClock needs to adjust for latencies of modules while scheduling notifications. To do so, it needs the latency value of all modules, and it also needs to know from which module the API is being called. One way of doing this is to assign ID values to all modules, which they can pass to media clock while calling APIs. Instead of creating IDs for modules, the media clock uses unique ClockNotificationsInterfaceImpl object for each module. Modules call APIs using their interface-object and the interface object calls corresponding APIs in the clock. The media clock then schedules notifications after adjusting for the latency.
PVMFMediaClockNotificationsInterface interface contains client side functions for setting notifications on media clock and PVMFMediaClockNotificationsImplInterface contains corresponding functions implemented by PVMFMediaClock.
PVMFMediaClockNotificationsInterface is implemented by ClockNotificationsInterfaceImpl class and PVMFMediaClockNotificationsImplInterface interface is implemented by PVMFMediaClock. There is a one to one mapping between the functions of these two interfaces. Functions implemented in ClockNotificationsInterfaceImpl class call the corresponding functions in PVMFMediaClock. The signature of the functions is same except for the object reference pointer that setCallBack functions in ClockNotificationsInterfaceImpl class pass to functions in PVMFMediaClock. For more details please refer to Section 3.7.

The PVMFMediaClock implements all functions of the above four interfaces. The following is the description of the other classes:

- OsclTimerObject – PVMFMediaClock inherits from OsclTimerObject to be able to implement active object.
- PVMFMediaClockObserver – When the media-clock uses count base timebase, it has to set itself as the clock observer of the timebase to receive count update notices. Therefore, the media clock implements this PVMFMediaClockObserver interface.
- PVMFMediaClockNotificationsObs – To be able to set callbacks on PVMFMediaClock, any class has to implement PVMFMediaClockNotificationsObs interface. The PVMFMediaClock itself implements this interface because it sets callbacks on itself for queuing NPT clock transition events.
- PVMFTimebase - A clock timebase may be specified by the user by using the SetClockTimebase() method. The specified timebase must implement the PVMFTimebase interface. The PVMFMediaClock uses the specified clock timebase as the basic time source (i.e., source of "ticks") to calculate the current clock value. The PVMFMediaClock itself implements PVMFTimebase interface so that it can act as a timebase for other instances of PVMFMediaClock. For more details on timebase please refer to Section 10

3.2. Clock Timebase

The PVMFMediaClock uses the specified clock timebase as the basic time source (i.e., source of "ticks") to calculate the current clock value. The specified timebase must implement the PVMFTimebase interface. Usually the clock timebase uses the system tickcount for the clock value so a clock timebase based on the OSCL tickcount is provided. The relationship between PVMFMediaClock, PVMFTimebase and user supplied timebase is depicted in the class diagram in Section 8 If no timebase is provided, PVMFMediaClock can still be used, but the clock time will not progress from the start time unless clock adjustments are made. PVMFMediaClock itself implements the PVMFTimebase interface so that it can act as a timebase for other instances of PVMFMediaClock.

The clock timebase for the PVMFMediaClock is expected to satisfy several requirements. The clock timebase must return the timebase value in microseconds and the value must be monotonically non-decreasing. The PVMFMediaClock estimates the current clock value using the timebase by subtracting the tickcount at the clock start time from the current tickcount to compute the latest value.

The PVMFMediaClock timing behavior can be modified by providing different timebases. By having a clock timebase returning the clock value X times "real-time", the PVMFMediaClock would report the time as running X times faster. For multimedia playback scenario, the playback speed could be adjusted this way. A timebase can also be a so-called "counted" timebase which increments based on asynchronous events that may not happen based on uniform time intervals (e.g., consider a case where video frames should advance based on user interaction – i.e., frame stepping).

3.3. State Machine

PVMFMediaClock maintains an internal state machine of 3 states: STOPPED, RUNNING, and PAUSED. The state transition diagram for the 3 states is shown below:

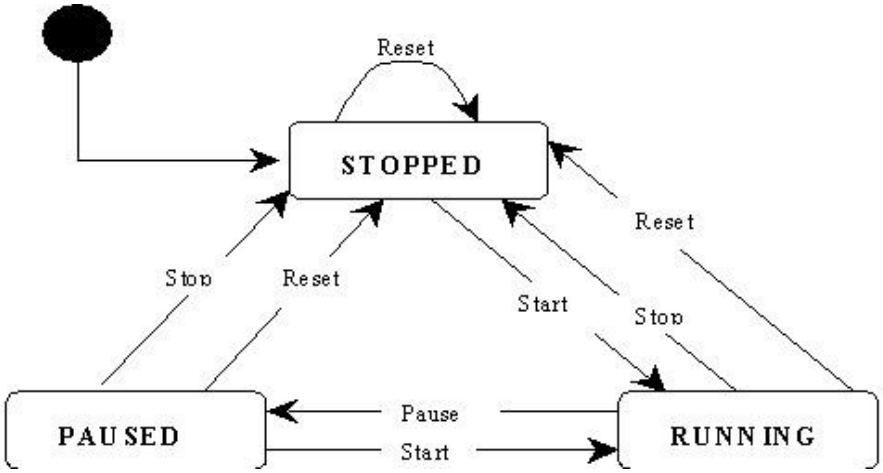

Figure 4: PVMFMediaClock State Diagram

When a PVMFMediaClock object is instantiated, it starts in the STOPPED state. While in STOPPED state, clock start time and the clock timebase can be set. When Start() is called, PVMFMediaClock transitions to the RUNNING state where the current clock time is sourced from

the clock timebase. While in RUNNING state, clock adjustments could be done via AdjustClockTime32() methods. AdjustClockTime32() allows the PVMFMediaClock time value to be adjusted to match another clock such as the audio output clock. The clock timebase cannot be changed while the clock is running. Also, ConstructMediaClockNotificationsInterface() function cannot be called while clock is running. When Pause() is called, PVMFMediaClock transitions to the PAUSED where the current clock time would freeze until the clock is started again. The clock time cannot be modified with AdjustClockTime32() methods in this state. The clock timebase can be changed while clock is in paused state. To return PVMFMediaClock to the RUNNING state from the PAUSED state, Start() needs to be called. The clock time then resumes from the point where it was paused and the clock time does not incorporate the timebase time elapsed while paused. When Stop() is called in RUNNING or PAUSED states, PVMFMediaClock returns to the initial STOPPED state and all runtime parameters such as last adjusted, start, and paused times are reset. Any active callbacks are fired with status code PVMFErrCallbackClockStopped and are deleted. Reset() can be called from any state of PVMFMediaClock. Calling Reset() puts the clock in STOPPED state and deletes all ClockObservers, ClockStateObservers and MediaClockNotificationsInterface objects. PVMFMediaClock does not send any notification to ClockObservers and ClockStateObservers before deleting them. Any active callbacks are fired with status code PVMFErrCallbackClockStopped when the clock is stopped at Reset(). The intent of the Reset() function is to bring the clock to the state when it was created.

3.4. Timekeeping

Internally, PVMFMediaClock stores all time values as unsigned 32-bit integer values. The default time unit for storing time is milliseconds but the clock switches to microseconds if user calls AdjustClockTime32() or SetStartTime32() with a microsecond time value. This is done to support modules that still use microseconds as time units. As the clock uses 32 bit variables to store time, there is a high probability that clock will wrap using microseconds as time units, but any components using the clock should be prepared for the time value to wrap. Time comparisons independent of the wrap point can be done using the time comparison utilities described in Section 15 The actual resolution of the clock would depend on the clock timebase resolution and the adjustment time resolutions.

The PVMFMediaClock uses 32 bit variables to store the tickcount value returned by timebase. There is a chance that a 32 bit variable would wrap during the lifetime of the clock as the tickcount may not start from zero when the clock starts. To overcome this problem, media clock stores the tickcount value at the start of clock and subtracts it from current tickcount before storing current tickcount value in the 32 bit variable. Using this mechanism, tickcount virtually starts from zero for the clock. Clock uses PVTimeComparisonUtils APIs to calculate the difference which takes care of the wrap-around scenario.

3.5. Clock Adjustments

In the RUNNING state, the PVMFMediaClock value can be adjusted to track another source of timing information by using the AdjustClockTime32() methods. To adjust the clock, information on the observed “desired” time must be provided along with the PVMFMediaClock time and timebase value at the moment of that observation. The “desired” time usually comes from another time source such as the audio device driver which controls the rate of playback or capture of audio samples. By adjusting the clock value based on observations of the audio device, the temporal synchronization of audio and video data, as well as any other media tracks, can be maintained.

In the AdjustClockTime32() method, the PVMFMediaClock timebase value at the time of the observation is provided to give an absolute scale for determining how recently the observation was made. The PVMFMediaClock has logic to avoid using obsolete information as part of the adjustment. If an observation is older than the time of the last adjustment, which is recorded in the PVMFMediaClock, then the observation is considered obsolete and is not used. Any values which have an observation time in the future (i.e., later than the current timebase value) are also not used (i.e., thrown out as invalid). In addition, the adjustment logic attempts to use the most recent observation so older observations may be thrown out. The diagram in helps illustrate some of these rules.

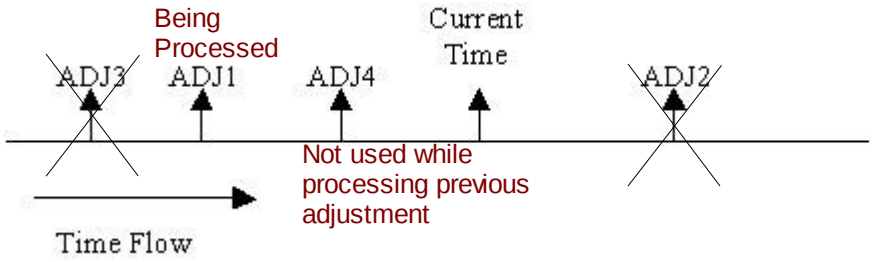

Figure 5: Restrictions on New Observations while Performing a Clock Adjustment.

The time is increasing from left to right in the diagram. At the “Current Time”, the clock is being adjusted with observations made at “ADJ1” (i.e., “ADJ1” is being processed). In this scenario, adjustment made with observations from “ADJ2” and “ADJ3” would not be allowed. “ADJ2” data is beyond the current time (i.e., in the future) and “ADJ3” data is older than “ADJ1” data. Adjustment with “ADJ4” is not excluded by the rules mentioned so far, but if there is already an adjustment being processed, then the default PVMFMediaClock algorithm will finish that processing and not utilize “ADJ4”. Supporting the interruption of a previous adjustment by another observation is more complicated so it was decided that the typical use-cases don't warrant that complication. Once the algorithm has completed processing “ADJ1”, the last adjustment time value is updated to reflect the timebase value at that moment. Any new observations must lie between the last adjustment time and the current time to be considered by the algorithm. Figure 6 illustrates the restriction where only “ADJ5” will be considered for new clock adjustments in that case.

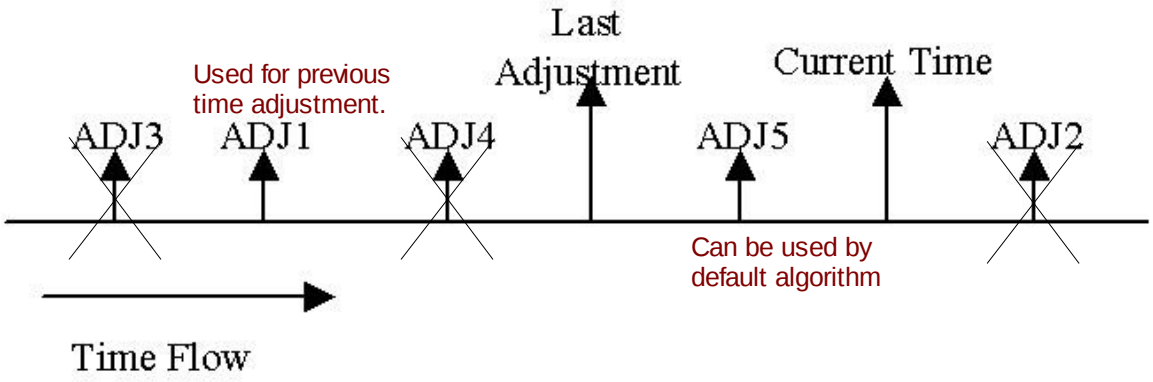

Figure 6: Restrictions on New Observations after Completing a Clock Adjustment.

To maintain the requirement that the time value should always be monotonically non-decreasing, calling AdjustClockTime32() with a desired adjusted clock value earlier than the PVMFMediaClock observed value will “freeze” the time value for a duration determined by the difference between two values according to the clock timebase. This default algorithm is implemented in the virtual function GetAdjustedRunningClockTime() so derived classes may override the function to use a different algorithm.

3.6. NPT Mapping

The NPT mapping is stored in the media clock in three variables. StartMediaClockTimestamp, StartNPT and clockDirection. These three values provide the necessary information for mapping the times during a continuous playback segment. When there is a discontinuous in jump in the NPT such as during repositioning or a change in direction, then the mapping needs to be updated. The method UpdateNPTClockPosition() provides a way to update these values. For the NPT mapping algorithm please refer to Section 5

3.7. Clock Notifications and Latency Handling

The PVMFMediaClock can notify components on changes in clock states or based on reaching specific values of the clock. For components that need to take action based on a specific value of the clock, the callback eliminates the need for setting their own timer and canceling and reseting it when the PVMFMediaClock pauses, restarts, or is adjusted. To get notifications for these events, modules have to implement corresponding observer interfaces. There are three such observer interfaces:

1.PVMFMediaClockObserver – Notified on clock Timebase update, clock count update and clock adjustment.

2.PVMFMediaClockStateObserver – Notified on clock state changes.

3.PVMFMediaClockNotificationsObs - These observers can set callbacks on clocks. For details on callbacks, please see Section 6

The PVMFMediaClock stores PVMFMediaClockObservers in a list. When a component adds itself as PVMFMediaClockObserver, the pointer supplied is added to the list. At the time of an event (timebase update, count update or clock adjustment), PVMFMediaClock goes through the list and calls appropriate function of the observer. Notifications for these observers are not adjusted for module latency.

Handling PVMFMediaClockStateObserver and PVMFMediaClockNotificationsObs observers is more complicated than PVMFMediaClockObserver observers. The reason for this is that PVMFMediaClock has to adjust for individual module latencies while scheduling notifications to these observers. For PVMFMediaClockStateObserver, only the clock-start notification is adjusted for module latency whereas clock-pause and clock-stop notifications are sent right away. All callback notifications to PVMFMediaClockNotificationsObs observers are adjusted for module latency.

The latency adjustment algorithm is as described in Section 7 To implement this method, the PVMFMediaClock needs the latency value of all modules to calculate largest common value and it also needs to know from which module the API is being called. To solve this, PVMFMediaClock uses unique PVMFMediaClockNotificationsInterfaceImpl objects for each module. Modules call the APIs using their interface object and the interface object calls corresponding APIs in the

media clock. The interface object adjusts for the latency in the time-value supplied to the clock. The following sequence diagram describes this mechanism.

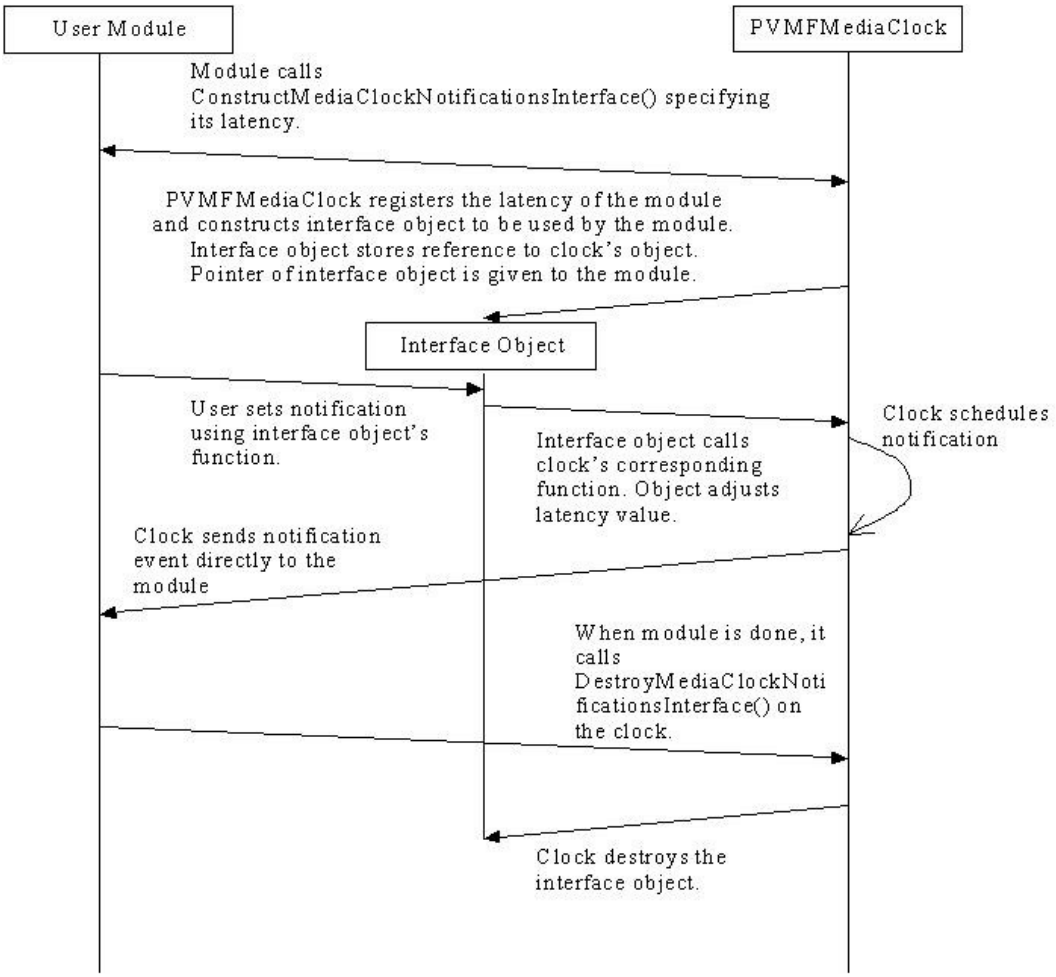

Figure 7: Sequence Diagram Showing Usage of PVMFMediaClockNotificationsInterface

3.8. Callbacks

Timer callbacks are implemented using a single active object. The PVMFMediaClock keeps all active timers in a priority queue with earlier expiring timers having the highest priority. It schedules the active object to the timer expiry time of the earliest expiring element in the queue. When a callback is fired, it is deleted from the queue and the AO is scheduled to expiry time of next pending timer in the queue. This AO is canceled when clock is paused/stopped and started again when clock starts.

While setting the callback timer, a module can give a time window for timer expiry rather than giving a specific time. When a timer at the front of the queue expires, next in line timers are checked to see if the current time is in their window or if they have expired. If it is, the next timer is fired too. Subsequent times are also checked until a timer is found whose window lies in the future. This logic works in a similar manner if window size is zero.

As described in the previous section, the callback timer APIs are accessed through a PVMFMediaClockNotificationsInterfaceImpl object. This object has to be explicitly created and destroyed. At the time of interface object destruction, if there are any active callback timers set through that interface object, they are deleted. For API usage example, please refer pvmf unit test code.

3.9. NPT Callbacks

NPT callbacks are implemented in the same way as regular callbacks. A separate priority queue is maintained for NPT timers.

3.10. PVMFMediaClockObservers and PVMFMediaClockStateObservers

The PVMFMediaClock stores PVMFMediaClockObservers in a list. When a module adds itself as a PVMFMediaClockObserver, the pointer supplied is added to the list. At the time of event (timebase update, count update, or clock adjustment), the PVMFMediaClock goes through the list and calls appropriate function of the objects.

PVMFMediaClockStateObservers are stored in the corresponding interface objects. When the clock state changes, the ClockStateUpdated function of all observers is called. The mechanism is different when the clock start notification has to be sent because the PVMFMediaClock has to adjust for latency while calling the PVMFMediaClockStateObservers for the clock start event. For all clock start notifications that have to be sent in future, the PVMFMediaClock sets a timer callback on itself. When these timers expire, the clock start notifications are sent.

3.11. NPT Clock Transition

User modules can set NPT clock transition events in the future. The PVMFMediaClock simply schedules the callback timers and changes the NPT clock when the callback expires. It stores the set of new values as an element in a queue and pops it when the time expires.

3.12. Multithreading Support

The PVMFMediaClock has limited multithreading support as of now. All regular and NPT Callback APIs have a threadlock flag. If this flag is set, class data is modified only after acquiring a common mutex.

4. Time Comparison Utilities

The time comparison utilities provide a small set of functions that take care common calculations needed when comparing time values. Although the time comparison utilities are not part of PVMFMediaClock itself, the implementation makes heavy use of them.

The time values are stored as 32-bit unsigned integers and will eventually wrap around the maximum integer value back to 0 given enough time. The time interval necessary to reach the wrap point depends on the initial starting value and the timescale (i.e, how quickly the time values increment). With the current default timescale of milliseconds, it would take 49+ days to reach the 32-bit limit starting from 0. However, if the timescale increases in the future to have higher resolution or it is not possible to guarantee a 0 initial time value, then the wrap point could be reached much quicker. Using simple integer comparisons to decide if one time value is earlier or later than another doesn't really work across the wrap point. A better way is to check the difference between two time values against a threshold. This method works even if the two values straddle the wrap point.

The diagram below shows a conceptual model of the time evolution of these values. The values progress in a clockwise direction around the circle as time advances. Once the value increments beyond the maximum 32-bit value, it wraps back to 0. The difference between two values can be seen by the distance between the two values in the counter-clockwise direction. For example, Figure 8 shows two values R0 and R1 where R1 occurs after (i.e., later than) R0 and the difference R1 - R0 is the distance between R1 and R0 in the counter-clockwise direction.

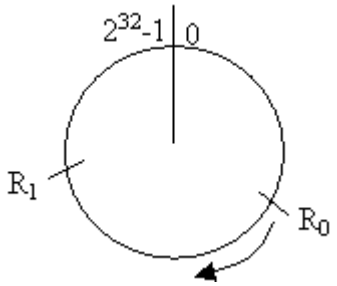

Figure 8: Conceptual Model of Time Values

The basic comparison utilities are placed in baselibs/media_data_structures along with the clock converter. The utilities are structured specifically for time comparisons. One function, called **IsEarlier()**, takes two values to compare as well as an output parameter to return the difference. The function determines if the first parameter is earlier than the second and always provides the difference as output. The difference is computed as part of the process so it is no extra work to return it, and since many callers need this information, the implementation included it.

The IsEarlier method can also be used to compare the stream IDs used the PVMFMediaMsg structure for carrying datapath messages because the design stipulates that the stream IDs must be assigned so the comparison shown above indicates the time ordering of two stream IDs.

Another utility is the CheckTimeWindow() function. This API can determine if a given timestamp falls within a time window, which is a common check used to decide if an event notification should be sent.