Digital Video Broadcasting (DVB); GEM Media Synchronization API

DVB Document A153

October 2010


DVB
Digital Video
Broadcasting

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for ETSI members and non-members, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://webapp.etsi.org/IPR/home.asp).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Specification (TS) has been produced by Joint Technical Committee (JTC) Broadcast of the European Broadcasting Union (EBU), Comité Européen de Normalisation ELECtrotechnique (CENELEC) and the European

NOTE: The EBU/ETSI JTC Broadcast was established in 1990 to co-ordinate the drafting of standards in the specific field of broadcasting and related fields. Since 1995 the JTC Broadcast became a tripartite body by including in the Memorandum of Understanding also CENELEC, which is responsible for the standardization of radio and television receivers. The EBU is a professional association of broadcasting organizations whose work includes the co-ordination of its members' activities in the technical, legal, programme-making and programme-exchange domains. The EBU has active members in about 60 countries in the European broadcasting area; its headquarters is in Geneva.

European Broadcasting Union
CH-1218 GRAND SACONNEX (Geneva)
Switzerland
Tel: +41 22 717 21 11
Fax: +41 22 717 24 81

The Digital Video Broadcasting Project (DVB) is an industry-led consortium of broadcasters, manufacturers, network operators, software developers, regulatory bodies, content owners and others committed to designing global standards for the delivery of digital television and data services. DVB fosters market driven solutions that meet the needs and economic circumstances of broadcast industry stakeholders and consumers. DVB standards cover all aspects of digital television from transmission through interfacing, conditional access and interactivity for digital video, audio and data. The consortium came together in 1993 to provide global standardisation, interoperability and future proof specifications.

Introduction

Purpose

The GEM Middleware Platform for interactive TV defines a technical solution for the user terminal that enables the reception and presentation of applications in a vendor, author and broadcaster neutral framework. Applications from various service providers are to be interoperable with different GEM implementations in a horizontal market, where applications, networks, and GEM terminals can be made available by independent providers.

In recent years TV-sets and Set-top boxes with more than one tuner have come to the market and hybrid Broadcast/Broadband TV sets are gaining a growing market share.

GEM and GEM-based terminal middleware specifications have been successfully deployed into hybrid markets and already address a broad spectrum of use-cases. The GEM specification already supports multi-tuner and multi-network devices today, but there are new use cases requiring a GEM extension for synchronization of media streams into a common service presentation.

Aim

The aim of the present document is to define a GEM extension to enable the synchronization of services over different networks to support advanced hybrid scenarios.

1 Scope

The present document is to be used by entities writing system specifications and/or standards that define a Java based TV middleware. The scope is to satisfy the DVB commercial requirements to enable smooth and full integration of services in a mixed broadcast (DVB-C/S/T, etc..) and broadband environment.

It defines extensions to the GEM specification family to enable synchronization of multiple media streams or stream components that may come into the terminal over different networks. The solution provides sufficient flexibility to enable:

- Synchronization of arbitrary numbers of media streams or stream components.
- Synchronization of streams over different networks as well as streams or stream components over multiple network interfaces to the same networks (e.g. Multi-tuner terminals).
- Synchronization of media streams or stream components in different formats.

The design of this extension is flexible enough to support also future GEM profiles and targets.

The present document enables a number of different deployment scenarios. These include devices with a dual or hybrid tuners as well as so-called “connected” terminals, consisting of a tuner and a DSL network connection for IPTV services.

The present document may be included in a future release of the GEM specification.

2 References

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references,only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication ETSI cannot guarantee their long term validity.

2.1 Normative references

The following referenced documents are necessary for the application of the present document.

[1] ETSI TS 102 728: "Digital Video Broadcasting (DVB); Globally Executable MHP (GEM) Specification 1.2.2".

[2] JSR 217: “Personal Basis Profile 1.1”

Note: At the time of writing, Personal Basis Profile 1.1.2 is the latest maintenance release available at http://jcp.org/en/jsr/summary?id=217. It is recommended to always use the latest release of this JSR.

[3] JSR 927: “Java TV™ API 1.1”

Note: At the time of writing, JavaTV 1.1.1 is the latest maintenance release available at http://jcp.org/en/jsr/summary?id=927. It is recommended to always use the latest release of this JSR.

2.2 Informative references

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[4] ETSI TS 102 727: "Digital Video Broadcasting (DVB); Multimedia Home Platform (MHP) Specification 1.2.2".

[5] CM 848r1: Commercial Requirements for Hybrid Broadcast/Broadband Services
DVB-CM IPTV group
DVB document CM848r1 / CM-IPTV0221r20 available at www.dvb.org

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the terms and definitions given in GEM [1] and the following apply:

HNED: Home Network Endpoint Device – a terminal device in the users home used for TV consumption, which may be connected to other home network devices. This could be a set-top box or an integrated TV set.

Hybrid Terminal: A HNED, which is connected to more than one network interface.

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

API	Application Program Interface, sometimes also referred to as Application Programming Interface
DVB	Digital Video Broadcasting
GEM	Globally Executable MHP as defined in GEM 1.2.2. [1].
HTTP	Hypertext Transfer Protocol
MHP	Multimedia Home Platform as defined in MHP 1.2.2. [4].

5 Use Cases (Informative)

This section describes the use cases and requirements that were defined by the IPTV group of the Commercial Module in the DVB, which are addressed by the API of the present document.

The following use cases have been considered (Source: CM848[5]):

1. Consumption of content from broadcast or broadband services where some content items may be available on both networks
2. Simultaneous consumption of independent broadcast and/or broadband services
3. Simultaneous consumption of content items where there is dependence between services or components in terms of synchronisation of presentation
4. Cross signalling of data only services between broadcast and broadband

Most of these use cases can be already addressed by the present GEM[1] specification:

GEM permits the combination of GEM profiles (=targets) into a combined profile in section 4.1.7, and thus enables the creation of hybrid terminals. There are already deployments of GEM-based hybrid terminals today.

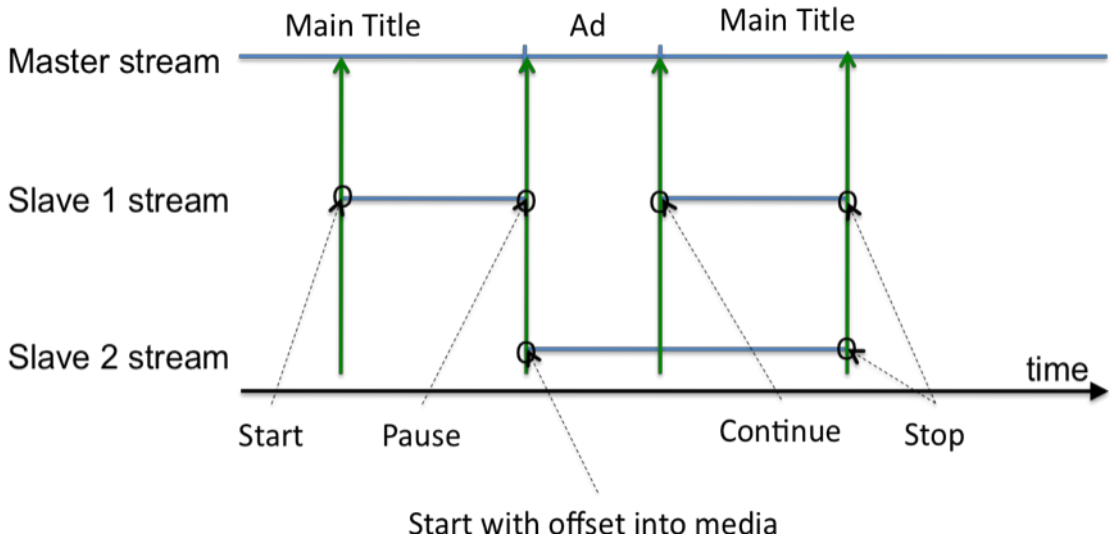
Receiving and aggregating metadata in the terminal to produce a coherent metadata set, which is the basis for all use cases above, is addressed in GEM, which defines mechanisms to combine service metadata from different networks into combined service information in Annex AO.

The ability to discover services and content from all of the access networks requires that metadata is available describing those services and content items. GEM offers combined service information via the JavaTV SI-APIs to which services and service metadata from different description formats (e.g. SD&S, BCG) are mapped.

GEM contains mechanisms to address and select the services from different networks. Assuming that the appropriate decoder capabilities exist on the terminal, GEM can already be used to render independent services from different networks simultaneously.

The ony use-case, which requires an extension of GEM is the “Simultaneous consumption of content items where there is dependence betweenbservices or components in terms of synchronisation of presentation”. This use case enables new scenarios, where streams over different network channels are synchronized and presented together. These scenarios include offering additional language choices or subtitle streams, providing an audio descriptor service for visually impaired people or delivering a sign language representation in addition to the service. Other scenarios are alternative endings to shows or themed adverts, which require a synchronized switching from one network to the other.

An example to illustrate the synchronization of 2 slave streams to a master stream is shown in Figure 1. In this example a slave stream 1, which could be a foreign language audio track is added to the presentation, while the master stream is already presented. It is synchronized and presented together with the master stream until the main title is interrupted by an ad break. When the ad starts, an additional second slave stream is started, which could be another synchronized media stream. When the main title continues, the first slave stream is resumed and the 3 streams continue to be presented until the two slave streams are stopped and the master stream is the only remaining stream. Note that the pausing and resuming of the slave streams could happen completely independent from the master stream under full control of an application.



- Figure 1: Example of synchronization of 3 media streams

6 Architecture

The synchronization API supports the system reference model for hybrid broadcast+IPTV delivery as defined by commercial requirements in the DVB (CM848[5]). This reference model describes a delivery network architecture, where coordinated head-ends deliver services to the home end point via separate delivery paths.

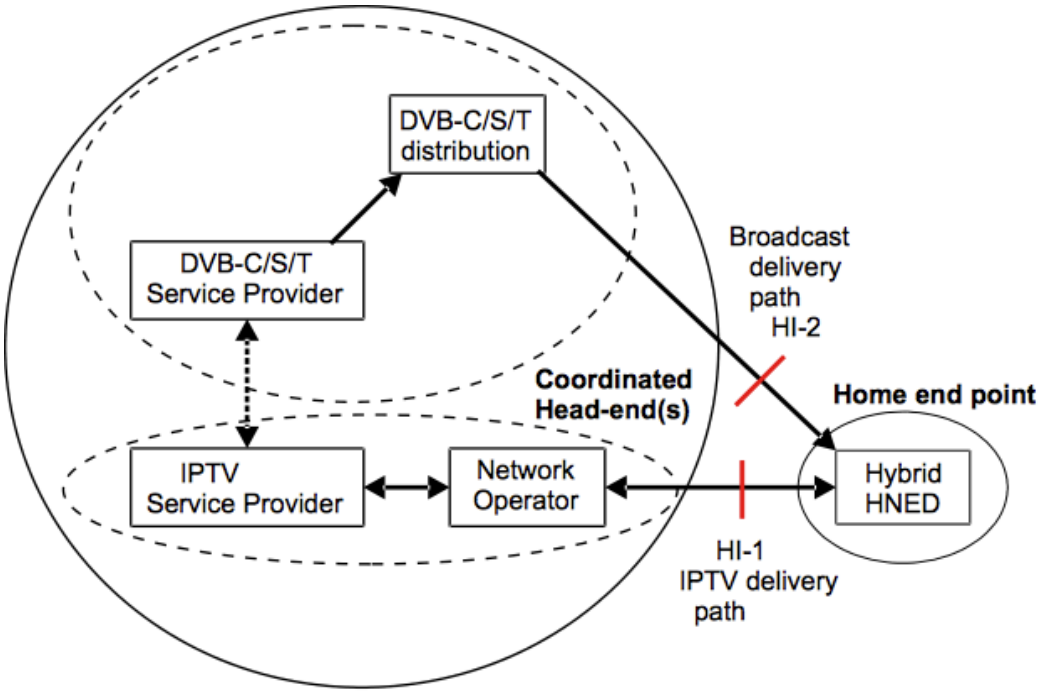


Figure 2: Hybrid Reference Model (Source: CM848[5])

The API defined in the present specification enables the synchronization of media streams that come into the terminal over different networks. To address the synchronization of streams on the HNED (= terminal device), a mechanism to enable the synchronization and combining of streams is defined. This is illustrated in the logical model in

Figure 3.

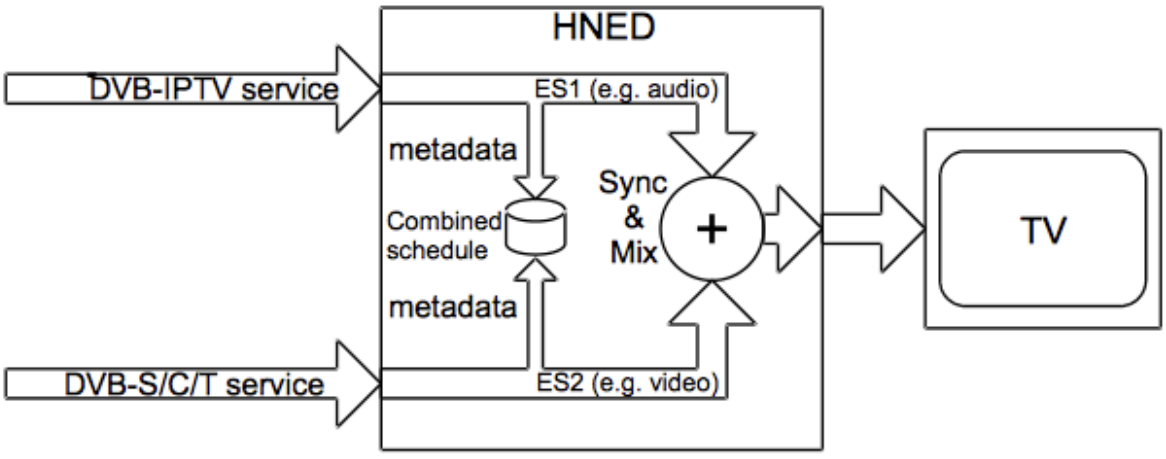


Figure 3: Logical model of a hybrid terminal (Source: CM848[5])

The extensions integrate into the GEM and JavaTV architecture and provide a new master/slave concept, which permits to have loosely coupled master slave players, where the slave player can still be controlled independently from the master.

This behavior is achieved with loosely coupled player state machines, which allows:

- Adding a slave to an already playing master.
- Stopping/restarting of a slave independently from the master.

The API maintains the “single player per datasource” design of JavaTV[, therefore there is a separate JMF player for every media stream on a different network.

The design of the API is flexible enough to enable:

- Synchronization of an arbitrary number of media streams.
- Media streams can be sent over different networks.
- Media streams can be in different content formats.
- Media streams can be sent over different protocols.

The synchronization API:

- Provides a new JMF control to establish a master/slave relationship between JMF players.
- Supports multiple masters/slaves per application.
- Contains an event system to notify the application about out-of-sync conditions.
- Provides a method to set the tolerance time value for permitted timing deviations during synchronization.

A synchronized service with streaming media service components, that may come into the terminal via different networks, is treated as a single service instance, which is handled from a single service context. After the service is selected in the service context, there is a dedicated ServiceMediaHandler for each network, i.e. there are more than one instances of JMF players per service context, which can be synchronized.

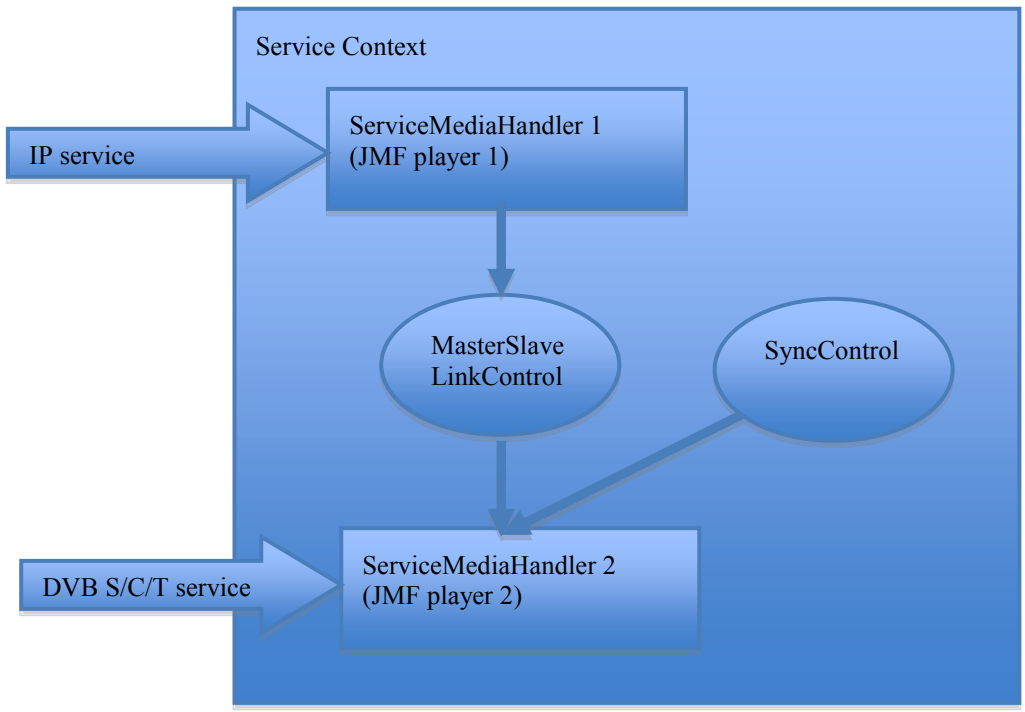


Figure 4: Master/Slave link between JMF players

The Synchronization API can be used to establish a Master/Slave link between a master JMF player and subordinate slave players. While the synchronization link is established, all slave players are synchronized to the master. Individual offsets can be used to set the slave players to different positions in their corresponding media streams.

The high level class diagram in Figure 5 shows the extensions of the GEM APIs, which define two new JMF controls: The MasterSlaveSyncLinkageControl to link between master and slave and a SyncControl to control the offset and tolerance of the

synchronized slave. Changes to the synchronization status, such as "in-sync" and "out of sync" conditions are communicated to SyncStateChangedEventListeners with SyncStateChangedEvents. The detailed API behavior is described in clause 7.

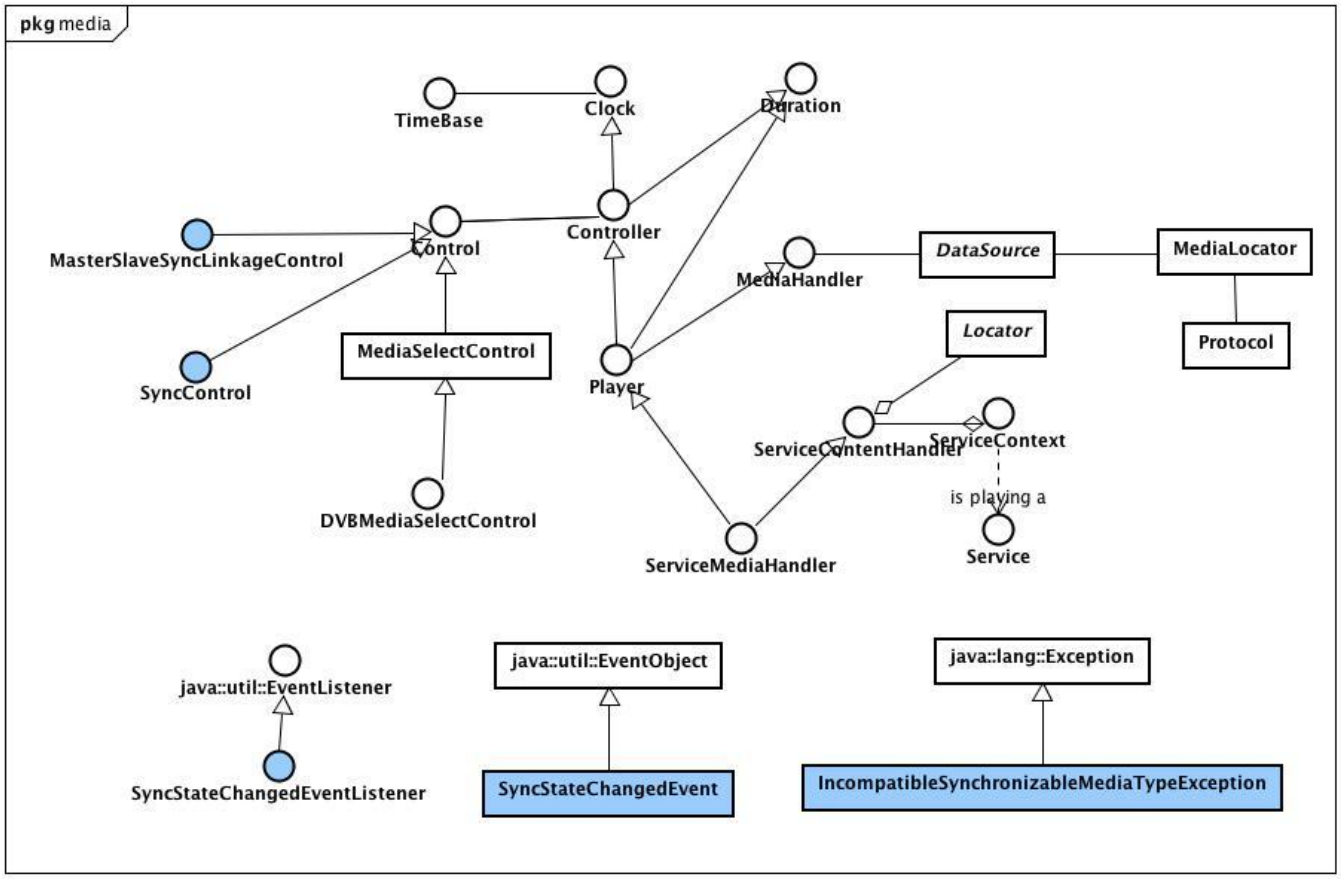


Figure 5: High level class diagram – JMF extensions

7 Stream Synchronization API (Normative)

The stream synchronization API is an optional extension of the GEM core APIs, which enables to establish a synchronization link between multiple JMF players. It enhances the JMF player model as defined in JavaTV [3], which already defines a limited master/slave concept (See the section "managing controllers" in class javax.media.Player in JavaTV for more details). For this purpose a set of new controls and events are defined, which provides a flexible master/slave extension which is in-line with the JavaTV architecture.

The stream synchronization API is defined in "Annex <A> (normative):
Stream Synchronization API"

Although this document provides an extension for GEM 1.2.2 [1], it could also be also applied to previous versions of GEM and derived terminal specifications.

The stream synchronization API can be deployed on any existing and future GEM targets. It may be used to synchronize streams on terminals deployed only on a single network (e.g. dual tuner STBs) as well as on hybrid terminals, on which the GEM targets are combined into a hybrid profile.

The stream synchronization API can be used to couple JMF players into a master and slave relationship and enables synchronization of the media presentation by synchronizing the clocks of these players. The media presentation can be controlled by controlling the master player, the slave player is synchronized accordingly. A master/slave relationship between JMF players lasts until the slave is explicitly removed from the master. Any JMF player becomes a master by adding a slave to it. Any JMF player becomes a slave, when it is added as a slave to another player.

Changes to the presentation state of the master player change the state of the slave player accordingly, i.e. starting a master player automatically starts the slave. Additionally all methods and controls, which change the media time or rate of the master player also have the same effect on the slave. Note that the slave can still be controlled directly, i.e. it can be moved to a different state (e.g. stopped) or set to a different media time. Any state change on a slave player doesn't affect the state of the master player. It is not permitted to set the media time and media rate of a slave player to a value different from the master player.

API behavior in border cases

If an application calls a method with a null parameter instead of an object instance, the method throws a NullPointerException. Methods which return an array of elements return an empty array if no element is available.

7.1 Establishing a Master/Slave Relationship

A master/slave relationship is a temporary synchronization relationship between JMF players. A player obtains the master role, i.e. it contains the reference clock to which other slave players can be synchronized. The master/slave relationship is established dynamically by adding a player as a slave to another player. It lasts until the slave is removed. The API permits to add multiple slaves to a master player. When a player has a slave role, it cannot act as a master for other players, i.e. there's only a one-level master/slave hierarchy.

An application, which wants to establish a master/slave relationship among players, can obtain an instance of `MasterSlaveSyncLinkageControl` of the master player. On platforms which don't allow synchronization, a call to `getControl("org.dvb.media.MasterSlaveSyncLinkageControl")` on any player will return null.

A master/slave relationship is established when another player is added to the master via the `addSlave` method of the `MasterSlaveSyncLinkageControl`, which becomes a slave player when the `addSlave` method was successfully executed. A SyncControl can be obtained for any player, which can be used as a slave player but the methods of `SyncControl` are only effective, while a Master/Slave relationship is established.

To remove a slave from a master/slave relationship, the method `removeSlave` is used. If a slave had been removed, it can be controlled independently from the master. After all slaves are removed from a master, the master is again a normal player and could serve as a slave to another master.

Adding or removing a slave player can happen in any state of the master and slave player. These operations don't impact the player states of the master and the slave, i.e. the return value of `Player.getState()` does not change for the master and slaves.

Not all media streams and their corresponding JMF players can be successfully synchronized - If synchronization can never be established during the `addSlave` method, an `IncompatibleSynchronizableMediaTypeException` is thrown.

7.1.1 Adding a Slave

When a slave has been added, the master `Player`:

- Invokes `setTimeBase` on the `slave` with the master `Player's TimeBase`. If this fails, `addSlave` throws an `IncompatibleTimeBaseException`.
- Synchronizes the slave with the `Player` using `setMediaTime`, `setStopTime`, and `setRate`.
- Takes the added slaves latency into account when computing the master `Player's` start latency. When `getStartLatency` is called, the master `Player` returns the greater of: its latency before the `Controller` was added and the latency of the added `Controller`.
- Takes the added slave's duration into account when computing the master `Player's` duration. When `getDuration` is called, the `Player` returns the greater of: its duration before the `slave` was added and the duration of the added `slave`. If either of these values is DURATION_UNKNOWN, `getDuration` returns DURATION_UNKNOWN. If either of these values is DURATION_UNBOUNDED `getDuration` returns DURATION_UNBOUNDED.
- Adds itself as a `ControllerListener` for the added slave so that it can manage the events that the slave generates. (See the Events section below for more information.)
- Invokes control methods on the added slave in response to methods invoked on the master `Player`. The methods that affect slave players are discussed below.

7.1.2 Removing a Slave

When a `slave` is removed from a master `Player's` control, the master `Player`:

- Resets the slave's `TimeBase` to its default.
- Recalculates its duration and posts a `DurationUpdateEvent` if the master `Player's` duration is changed.
- Recalculates its start latency.

7.2 Starting a Master Player

In accordance with the behavior for managed players as described in JavaTV, the start behavior is as follows:

When you call `start` on a master `Player`, all of the `slaves` managed by the `Player` are transitioned to the *Prefetched* state. When the `slaves` are *Prefetched*, the master `Player` calls `syncStart` with a time consistent with the latencies of each of the slaves.

Calling realize, prefetch, stop, or deallocate on a Master Player
When you call `realize`, `prefetch`, `stop`, or `deallocate` on a master `Player`, the `Player` calls that method on all of the `slaves` that it is managing. The `Player` moves from one state to the next when all of its `slaves` have reached that state. For example, a `Player` in the *Prefetching* state does not transition into the *Prefetched* state until all of its `slaves` are *Prefetched*. The `Player` posts `TransitionEvents` normally as it changes state.

Calling syncStart or setStopTime on a Master Player
When you call `syncStart` or `setStopTime` on a master `Player`, the `Player` calls that method on all of the `slaves` that it is managing. (The `Player` must be in the correct state or an error is thrown. For example, the `Player` must be *Prefetched* before you can call `syncStart`.)

Setting the Time Base of a Master Player

When setTimeBase is called on a master Player, the Player calls setTimeBase on all of the slaves it's managing. If setTimeBase fails on any of the Controllers, an IncompatibleTimeBaseException is thrown and the TimeBase last used is restored for all of the Controllers.

Getting the Duration of a Master Player
Calling getDuration on a master Player returns the maximum duration of all of the added Controllers and the master Player. If the Player or any slave has not resolved its duration, getDuration returns Duration.DURATION_UNKNOWN.

Closing a Master Player
When close is called on a master Player all slaves are closed as well.

7.3 Setting the Media Time and Rate of a Master Player

In accordance with the behavior for managed players as described in JavaTV, the event handling is as follows:

When you call setMediaTime on a master Player, its actions differ depending on whether or not the Player is *Started*. If the master Player is not *Started*, it simply invokes setMediaTime on all of the slaves it's managing. If the Player is *Started*, it posts a RestartingEvent and performs the following tasks for each slave:

- Invokes stop on the slave.
- Invokes setMediaTime on the slave.
- Invokes prefetch on the slave.
- Waits for a PrefetchCompleteEvent from the slave.
- Invokes syncStart on the slave

The same is true when setRate is called on a master Player. The Player attempts to set the specified rate on all slaves, stopping and restarting the Controllers if necessary. If some of the Controllers do not support the requested rate, the Player returns the rate that was actually set. All Controllers are guaranteed to have been successfully set to the rate returned.

7.4 Loss of Synchronization

The API permits to set a synchronization tolerance time value, within which the different Media streams may deviate from the master. If the master/slave player cannot keep the players synchronized within the tolerance value, this condition is reported with a SyncStateChangedEvent. After the synchronization is re-established between the master and the slave, this is also reported with a SyncStateChangedEvent.

A slave player always attempts to resynchronize to the master, however when it determines that synchronization will never be established, it set its status to unsynchronizable and reports this with an SyncStateChangedEvent with unsynchronzable.

7.5 Event Handling

In accordance with the event behavior for managed players as described in JavaTV, the event handling is as follows:

Most events posted by a slave are filtered by the master Player. Certain events are sent directly from the slave through the master and to the listeners registered with the master.
To handle the events that a slave can generate, the master registers a listener with the slave when the master/slave relationship is established. Other listeners that are registered with the slave are still notified, while it is being managed by the master. When a slave is removed from the master Player's list of slaves, the slave removes itself from the masters listener list.

Transition Events

A master Player posts TransitionEvents normally as it moves between states, but the slaves affect when the

master changes state. In general, a master `Player` does not post a transition event until all of its slaves have posted the event.

Status Change Events

The master `Player` collects the `RateChangeEvents`, `StopTimeChangeEvents`, and `MediaTimeSetEvents` posted by its slaves and posts a single event for the group.

DurationUpdateEvent
A JMF `Player` posts a `DurationUpdateEvent` when it determines its duration or its duration changes. A master `Player's` duration might change if a slave updates or discovers its duration. In general, if a slave posts a `DurationUpdateEvent` and the new duration changes the master's duration, the master Player posts a `DurationUpdateEvent`

CachingControlEvent
A master `Player` reposts `CachingControlEvents` received from a `Players` that it manages, but otherwise ignores the events.

ControllerErrorEvents
A master `Player` immediately reposts any `ControllerErrorEvent` received from a `slave` that it is managing to its `ControllerListener`. After a `ControllerErrorEvent` has been received from a slave, a master `Player` no longer invokes any methods on the slave; the slave is ignored from that point on and the slave is removed from the master.

Annex <A> (normative):
Stream Synchronization API

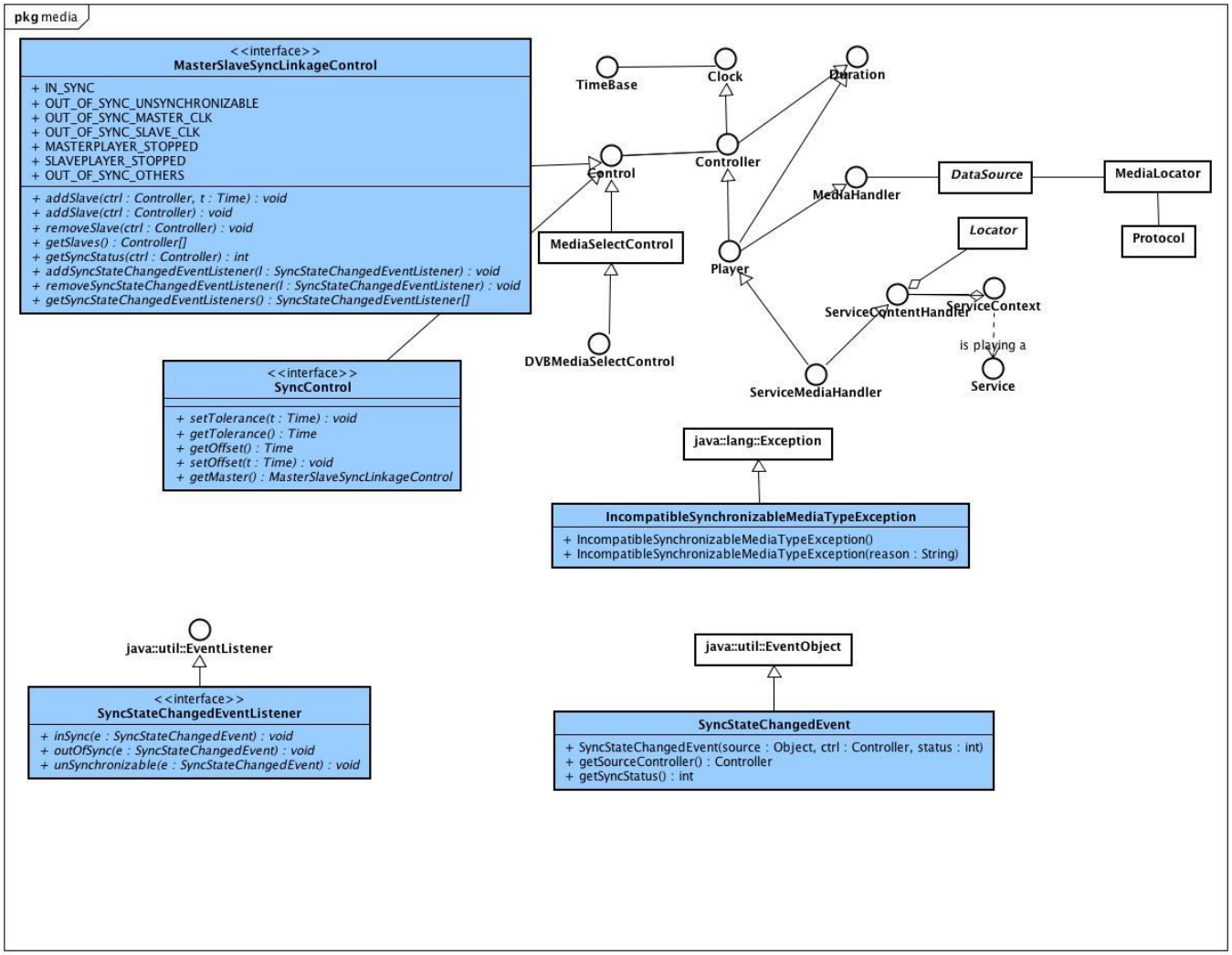


Figure 6: Stream Synchronization API

Package org.dvb.media

Interface Summary		Page
MasterSlaveSyncLinkageControl	MasterSlaveSyncLinkageControl handles Player-based Master/Slave relationship for synchronization of JMF Players.	*35*
SyncControl	SyncControl is a Control object for synchronization of a slave JMF Player.	*35*
SyncStateChangedEventListener	This listener interface is for receiving synchronization change events among JMF Players.	*35*

Class Summary		Page
SyncStateChangedEvent	An event which indicates change of status of synchronization between master/slave JMF Players.	*35*

Exception Summary		Page
IncompatibleSynchronizableMediaTypeException	This exception is thrown when media type of each player is incompatible for synchronization.	*35*

Class IncompatibleSynchronizableMediaTypeException

org.dvb.media

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └org.dvb.media.IncompatibleSynchronizableMediaTypeException
```

All Implemented Interfaces:

Serializable

```
public class IncompatibleSynchronizableMediaTypeException
extends Exception
```

This exception is thrown when media type of each player is incompatible for synchronization.

Constructor Summary	*Page*
IncompatibleSynchronizableMediaTypeException() Constructs IncompatibleSynchronizableMediaTypeException.	*35*
IncompatibleSynchronizableMediaTypeException(String reason) Constructs IncompatibleSynchronizableMediaTypeException with reason.	*35*

Constructor Detail

IncompatibleSynchronizableMediaTypeException

```
public IncompatibleSynchronizableMediaTypeException()
```

Constructs IncompatibleSynchronizableMediaTypeException.

IncompatibleSynchronizableMediaTypeException

```
public IncompatibleSynchronizableMediaTypeException(String reason)
```

Constructs IncompatibleSynchronizableMediaTypeException with reason.

Parameters:

`reason` - String object describing the reason of this exception.

Interface MasterSlaveSyncLinkageControl

org.dvb.media

All Superinterfaces:
Control

```
public interface MasterSlaveSyncLinkageControl
extends Control
```

MasterSlaveSyncLinkageControl handles Player-based Master/Slave relationship for synchronization of JMF Players. A JMF Player that provides master clock for synchronization is a *master Player*. MasterSlaveSyncLinkageControl allows a JMF Player from which this Control is obtained to be a master Player. At the time MasterSlaveSyncLinkageControl object is obtained from a Controller object of a JMF Player, the Player is a potential master Player. When a Controller object of a *slave Player* is added this control by `addSlave()` method, actual synchronization relationship is established. If synchronization between the Players is successfully established, a JMF Player to which MasterSlaveSyncLinkageControl object belongs becomes a master Player. When all the slave Players are removed from the MasterSlaveSyncLinkageControl object by `removeSlave()` method, a master Player is back to a potential master Player.

MasterSlaveSyncLinkageControl object monitors synchronization status between the master and slave Players. When synchronization status is changed, MasterSlaveSyncLinkageControl object notifies the situation by throwing SyncStateChangedEvent to the application. Before adding a slave Player, synchronization status is 'Unsynchronized'. At the time a slave Player is registered by `addSlave()` method, synchronization status is 'Synchronizing'. When synchronization is successfully established, the status is 'Synchronized'. In case that established synchronization cannot be kept within given tolerance, the status returns to 'Synchronizing' and related control objects try to re-synchronize automatically unless synchronization status is set to *OUT_OF_SYNC_UNSYNCHRONIZABLE*.

MasterSlaveSyncLinkageControl supports only simple Master/Slave relationship. That is, the object accepts relationship between a single master Player and slave Players. It is not allowed that a single JMF Player becomes a master Player and a slave Player at the same time. (A potential master Player can be a slave Player.) In addition, a slave Player can have only a single master Player. In case that an application tries to establish the relationship that violates these conditions, *IllegalArgumentException* is thrown when `addSlave()` is called.

Instance of MasterSlaveSyncLinkageControl can be obtained from a JMF Controller via the methods getControl(String) and getControls(). A single Controller object of a JMF Player can create at most one instance of MasterSlaveSyncLinkageControl.

Synchronization Status and Status Constants

Unsynchronized	Synchronizing	Synchronized
Master/Slave relationship is not established and Players can be running freely.	*MASTERPLAYER_STOPPED*, *SLAVEPLAYER_STOPPED*, *OUT_OF_SYNC_MASTER_CLK*, *OUT_OF_SYNC_SLAVE_CLK*, *OUT_OF_SYNC_OTHERSs*	*IN_SYNC*
		Unsynchronizable OUT_OF_SYNC_UNSYNCHRONIZABLE

Field Summary		Page
int	**IN_SYNC** Status representing synchronized	35

int	**MASTERPLAYER_STOPPED** Status representing stop() method is applied to a master Player	*35*
int	**OUT_OF_SYNC_MASTER_CLK** Status representing out of sync due to unstable clock of master Player	*35*
int	**OUT_OF_SYNC_OTHERS** Status representing out of sync due to other reasons	*35*
int	**OUT_OF_SYNC_SLAVE_CLK** Status representing out of sync due to unstable clock of slave Player	*35*
int	**OUT_OF_SYNC_UNSYNCHRONIZABLE** Status representing out of sync and Unsynchronzable state	*35*
int	**SLAVEPLAYER_STOPPED** Status representing stop() method is applied to a slave Player	*35*

Method Summary		*Page*
void	**addSlave**(Controller ctrl) Adds a specified Player as a slave Player that synchronizes with a master Player.	*35*
void	**addSlave**(Controller ctrl, Time t) Adds a specified Player as a slave Player that synchronizes with a master Player.	*35*
void	**addSyncStateChangedEventListener**(SyncStateChangedEventListener l) Adds the specified SyncStateChangedEventListener to receive the events.	*35*
Controller[]	**getSlaves**() Gets slave Players added by addSlave method.	*35*
SyncStateChangedEventListener[]	**getSyncStateChangedEventListeners**() Returns an array of all the SyncStateChangedEvent listeners registered on this control.	*35*
int	**getSyncStatus**(Controller ctrl) Gets currrent synchronization status of s specified slave Player.	*35*
void	**removeSlave**(Controller ctrl) Removes a specified slave Player.	*35*
void	**removeSyncStateChangedEventListener**(SyncStateChangedEventListener l) Removes a specified SyncStateChangedEventListener to receive the events.	*35*

Field Detail

IN_SYNC

```
public static final int IN_SYNC
```

Status representing synchronized

OUT_OF_SYNC_UNSYNCHRONIZABLE

```
public static final int OUT_OF_SYNC_UNSYNCHRONIZABLE
```

Status representing out of sync and Unsynchronzable state

OUT_OF_SYNC_MASTER_CLK

```
public static final int OUT_OF_SYNC_MASTER_CLK
```

Status representing out of sync due to unstable clock of master Player

OUT_OF_SYNC_SLAVE_CLK

```
public static final int OUT_OF_SYNC_SLAVE_CLK
```

Status representing out of sync due to unstable clock of slave Player

MASTERPLAYER_STOPPED

```
public static final int MASTERPLAYER_STOPPED
```

Status representing stop() method is applied to a master Player

SLAVEPLAYER_STOPPED

```
public static final int SLAVEPLAYER_STOPPED
```

Status representing stop() method is applied to a slave Player

OUT_OF_SYNC_OTHERS

```
public static final int OUT_OF_SYNC_OTHERS
```

Status representing out of sync due to other reasons

Method Detail

addSlave

```
void addSlave(Controller ctrl,
              Time t)
       throws InsufficientResourcesException,
              IncompatibleSynchronizableMediaTypeException,
              IllegalArgumentException
              IncompatibleTimeBaseException
```

Adds a specified Player as a slave Player that synchronizes with a master Player. Offset value given as a Time object states offset of time position of the specified slave Player and the master Player. If the specified Player already establishes synchronization, this method does nothing.

Parameters:

`ctrl` - Controller object of a potential slave Player to synchronize with this master Player.

`t` - Time object to represent offset between master Player's clock and the specified Player.

Throws:

`InsufficientResourcesException` - If the operation cannot be completed due to a lack of system resources.

`IncompatibleSynchronizableMediaTypeException` - If media type Player handles is incompatible to establish synchronization.

`IllegalArgumentException` - If a specified Controller object already forms a master or slave Player.

`IncompatibleTimeBaseException` - If invocation of `setTimeBase` on the adding slave Player with the master Player's TimeBase fails while establishing synchronization.

---

addSlave

```
void addSlave(Controller ctrl)
       throws InsufficientResourcesException,
              IncompatibleSynchronizableMediaTypeException,
              IllegalArgumentException
              IncompatibleTimeBaseException
```

Adds a specified Player as a slave Player that synchronizes with a master Player. Offset value between master Player and a specified Player is implicitly given as zero in this method. If the specified Player already establishes synchronization, this method does nothing.

Parameters:

`ctrl` - Controller object of a potential slave Player to synchronize with this master Player.

Throws:

`InsufficientResourcesException` - If the operation cannot be completed due to a lack of system resources.

`IncompatibleSynchronizableMediaTypeException` - If media type Player handles is incompatible to establish synchronization.

`IllegalArgumentException` - If a specified Controller object already forms a master or slave Player.

`IncompatibleTimeBaseException` - If invocation of `setTimeBase` on the adding slave Player with the master Player's TimeBase fails while establishing synchronization.

---

removeSlave

```
void removeSlave(Controller ctrl)
          throws IllegalArgumentException
```

Removes a specified slave Player. The specified Player turns self-running. This removal should not affect any progress of a master Player.

Parameters:

`ctrl` - Controller object of a potential slave Player to synchronize with this master Player.

Throws:

`IllegalArgumentException` - If the specified Controller object is not registered as a slave Player.

---

getSlaves

```
Controller[] getSlaves()
```

Gets slave Players added by addSlave method. Actual synchronization status of each slave Player is not considered in return value of this method.

Returns:

An array of Controller objects which are added by addSlave method.

---

getSyncStatus

```
int getSyncStatus(Controller ctrl)
```

Gets currrent synchronization status of s specified slave Player.

Parameters:

`ctrl` - Controller objcet of a slave Player object to obtain current synchronization status.

Returns:

Synchronization status in field values.

See Also:

Fields

---

addSyncStateChangedEventListener

`void` **`addSyncStateChangedEventListener`**`(`SyncStateChangedEventListener `l)`

Adds the specified SyncStateChangedEventListener to receive the events. SyncStateChangedEvent occurs when synchronization status between the master Player and any slave Player is changed. If l is null, no exception is thrown and no action is performed.

Parameters:

`l` - SyncStateChangedEvent listener.

---

removeSyncStateChangedEventListener

`void` **`removeSyncStateChangedEventListener`**`(`SyncStateChangedEventListener `l)`

Removes a specified SyncStateChangedEventListener to receive the events. If l is null, no exception is thrown and no action is performed.

Parameters:

`l` - SyncStateChangedEvent listener.

---

getSyncStateChangedEventListeners

SyncStateChangedEventListener`[]` **`getSyncStateChangedEventListeners`**`()`

Returns an array of all the SyncStateChangedEvent listeners registered on this control.

Returns:

An array of registered SyncStateChangedEvent listeners.

Interface SyncControl

org.dvb.media

All Superinterfaces:

Control

```
public interface SyncControl
extends Control
```

SyncControl is a Control object for synchronization of a slave JMF Player. SyncControl manipulates offset in time with a master Player and allowable torelance of synchronization. When SyncControl object is obtained from a Controller object of a JMF Player by getControl(String) or getControls() methods, the JMF Player is a potential slave Player. Such a JMF Player becomes a slave Player when the Player is registered as a slave Player by `addSlave()` methods of MasterSlaveSyncLinkageControl.

SyncControl supports only simple Master/Slave relationship. That is, the object accepts relationship between a single master Player and slave Players. It is not allowed that a single JMF Player becomes a master Player and a slave Player at the same time. In addition, a slave Player can have only a single master Player.

Instance of SyncControl can be obtained from a JMF Player via the methods getControl(String) and getControls(). A single JMF Player can create at most one instance of SyncControl.

Method Summary		*Page*
MasterSlaveSyncLinkageControl	**getMaster**() Gets a MasterSlaveSyncLinkageControl object that this SyncControl is currently synchronizing with.	*35*
Time	**getOffset**() Gets offset value of a specified slave Player against master Player, by Time object.	*35*
Time	**getTolerance**() Gets assigned tolerance value of synchronization for a specified slave Player.	*35*
void	**setOffset**(Time t) Sets offset value in time for a slave Player to the master Player.	*35*
void	**setTolerance**(Time t) Sets threshold to determine whether or not a specified slave Player establishes synchronization.	*35*

Method Detail

setTolerance

```
void setTolerance(Time t)
```

Sets threshold to determine whether or not a specified slave Player establishes synchronization. In other words, an application allows error of synchronization within a given parameter, t.

Parameters:

`t` - Allowable tolerance of synchronization in terms of Time object.

---

getTolerance

`Time` **`getTolerance`**`()`

Gets assigned tolerance value of synchronization for a specified slave Player.

Returns:

Value of tolerance for synchronization in terms of Time object.

---

getOffset

`Time` **`getOffset`**`()`

Gets offset value of a specified slave Player against master Player, by Time object.

Returns:

Offset value in terms of Time object.

---

setOffset

`void` **`setOffset`**`(Time t)`

Sets offset value in time for a slave Player to the master Player. The value is given by Time object.

Parameters:

`t` - Offset value in terms of Time object.

---

getMaster

`MasterSlaveSyncLinkageControl` **`getMaster`**`()`

Gets a MasterSlaveSinkLinkageControl object that this SyncControl is currently synchronizing with.

Returns:

MasterSlaveSinkLinkageControl object that this SyncControl is currently synchronizing with.

Class SyncStateChangedEvent

org.dvb.media

```
java.lang.Object
  └java.util.EventObject
      └org.dvb.media.SyncStateChangedEvent
```

All Implemented Interfaces:

Serializable

```
public class SyncStateChangedEvent
extends EventObject
```

An event which indicates change of status of synchronization between master/slave JMF Players.

This event is generated by MasterSlaveSyncLinkageControl object when status of synchronization between any combination of the master Player and slave Players.

Constructor Summary	*Page*
SyncStateChangedEvent(Object source, Controller ctrl, int status) Constructs a SyncStateChangedEvent object.	*35*

Method Summary		*Page*
Controller	**getSourceController**() Gets Controller object of a slave JMF Player object that is related to this event.	*35*
int	**getSyncStatus**() Gets changed synchronization status.	*35*

Constructor Detail

SyncStateChangedEvent

```
public SyncStateChangedEvent(Object source,
                             Controller ctrl,
                             int status)
```

Constructs a SyncStateChangedEvent object.

Parameters:

`ctrl` - Controller object of a slave Player that is related to this object.

`status` - current status of synchronization as a result of its change.

See Also:

Fields in SyncControl

Method Detail

getSourceController

```
public Controller getSourceController()
```

Gets Controller object of a slave JMF Player object that is related to this event.

Returns:

A slave Player object related to this event object.

---

getSyncStatus

```
public int getSyncStatus()
```

Gets changed synchronization status.

Returns:

changed synchronization status.

Interface SyncStateChangedEventListener

org.dvb.media

All Superinterfaces:

EventListener

```
public interface SyncStateChangedEventListener
extends EventListener
```

This listener interface is for receiving synchronization change events among JMF Players. The class that is interested in management of synchronization among JMF Players is registered with MasterSlaveSyncLinkageControl object using addSyncStateChangedEventListener method. The event is generated when synchronization status among JMF Players is changed. The relevant method in the listener object is then invoked, and the SyncStateChangedEvent is passed to it.

Method Summary		*Page*
void	**inSync**(SyncStateChangedEvent e) Invoked when any combination of master and slave Players are synchronized.	*35*
void	**outOfSync**(SyncStateChangedEvent e) Invoked when any combination of master and slave Players are desynchronized.	*35*
void	**unSynchronizable**(SyncStateChangedEvent e) Invoked when synchronization is never established.	*35*

Method Detail

inSync

```
void inSync(SyncStateChangedEvent e)
```

Invoked when any combination of master and slave Players are synchronized.

---

outOfSync

```
void outOfSync(SyncStateChangedEvent e)
```

Invoked when any combination of master and slave Players are desynchronized.

---

unSynchronizable

```
void unSynchronizable(SyncStateChangedEvent e)
```

Invoked when synchronization is never established.

Annex <B> (informative): Sample Code

```
import java.io.IOException;
import javax.media.Controller;
import javax.media.Manager;
import javax.media.NoPlayerException;
import javax.media.Player;
import javax.media.Time;
import javax.tv.service.selection.InsufficientResourcesException;

import org.davic.media.MediaLocator;
import org.davic.net.InvalidLocatorException;
import org.davic.net.dvb.DvbLocator;
import org.dvb.media.IncompatibleSynchronizableMediaTypeException;
import org.dvb.media.MasterSlaveSyncLinkageControl;
import org.dvb.media.SyncControl;
import org.dvb.media.SyncStateChangedEvent;
import org.dvb.media.SyncStateChangedEventListener;
import org.havi.ui.HPermissionDeniedException;
import org.havi.ui.HScreen;
import org.havi.ui.HVideoDevice;


public class SyncControlExample implements SyncStateChangedEventListener {

     private Player broadcastPlayer = null;
     private Player CCPlayer = null;
     private org.davic.media.MediaLocator CCMediaLocator;
     private DvbLocator CCLocator;
     private SyncControl bpSC = null;
     private SyncControl cpSC = null;
     private MasterSlaveSyncLinkageControl aMSSLC = null;

     public SyncControlExample() {

      // Create Broadcast Player
      HScreen screen = HScreen.getDefaultHScreen();
      HVideoDevice vd = screen.getHVideoDevices()[0];

      try {
         broadcastPlayer = (Player)vd.getVideoController();
      } catch (SecurityException e) {
         e.printStackTrace();
      } catch (HPermissionDeniedException e) {
         e.printStackTrace();
      }
      // Obtain SyncControl for Broadcast Player to be a slave Player
      bpSC = (SyncControl)broadcastPlayer.getControl("org.dvb.media.SyncControl");

      // Obtain MasterSlaveSyncLinkageControl to be a master Player
      aMSSLC =
(MasterSlaveSyncLinkageControl)broadcastPlayer.getControl("org.dvb.media.MasterSlaveSyncLinkageContr
ol");

      // Add listener for event related to synchronization
      aMSSLC.addSyncStateChangedEventListener(this);

      // Instantiate closed caption Player
      try {
         CCLocator = new DvbLocator("http://service/CCcomponent");
      } catch (InvalidLocatorException e2) {
         e2.printStackTrace();
      }
      CCMediaLocator = new MediaLocator(CCLocator);
      try {
         CCPlayer = Manager.createPlayer(CCMediaLocator);
      } catch (NoPlayerException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }

      // Obtain SyncControl to be a slave Player
      cpSC = (SyncControl)CCPlayer.getControl("org.dvb.media.SyncControl");

      if (CCPlayer != null) CCPlayer.start();

      Time tolerance = new Time(0.03); // Tolerance = 30ms
      Time offset = new Time(3.3); // Offset = 3.3s

      cpSC.setTolerance(tolerance);

      // Add closed caption player as a slave Player
      // Synchronization process starts automatically.
      try {
```

```
            aMSSLC.addSlave(CCPlayer, offset);
         } catch (InsufficientResourcesException e) {
            e.printStackTrace();
         } catch (IllegalArgumentException e) {
            // Checking reasons
            // Followings are sample to check whether CCPlayer is already a slave Player of other
master
            if (cpSC.getMaster() != null) {
               aMSSLC.removeSlave(CCPlayer);
            }
            // Followings are sample to check whether CCPlayer is a master
            // Instance of SyncControl can be created at most one.
            if
(((MasterSlaveSyncLinkageControl)CCPlayer.getControl("org.dvb.media.MasterSlaveSyncLinkageControl"))
.getSlaves().length != 0) {
               aMSSLC.removeSlave(CCPlayer);
            }
            e.printStackTrace();
         } catch (IncompatibleSynchronizableMediaTypeException e) {
            e.printStackTrace();
         } catch (IncompatibleTimeBaseException e) {
            e.printStackTrace();
         }

      }

      /*
       * Methods as an SyncStateChangedEvent event listener
       */

      public void inSync(SyncStateChangedEvent e) {

      }

      public void outOfSync(SyncStateChangedEvent e) {
       System.out.println("Now Resyncing...");
       Controller ctrl = e.getSourceController();
       int status = e.getSyncStatus();
       switch(status) {
       case MasterSlaveSyncLinkageControl.OUT_OF_SYNC_MASTER_CLK : {
          do_something;
          break;
       }
       case MasterSlaveSyncLinkageControl.OUT_OF_SYNC_SLAVE_CLK : {
          do_another_thing;
          break;
       }
        ....
       }

      }

      public void unSynchronizable(SyncStateChangedEvent e) {
       MasterSlaveSyncLinkageControl aSC = (MasterSlaveSyncLinkageControl)e.getSource();
       Controller aCtrl = e.getSourceController();
       aSC.removeSlave(aCtrl);  // Remove a slave Player that 'Give-up' to synchronize

      }
}
```

History

Document history		
<Version>	<Date>	<Milestone>
V0.1	Aug 2010	Initial draft
V0.2	12 Aug 2010	Updated API description, JavaDoc, Introduction
V0.3	19 Aug 2010	Updated Architecture, API description, Revised JavaDoc, Sample code
V0.4	25 Aug 2010	Revised JavaDoc + Sample code
V0.5	14 Sep 2010	Minor editorials
V0.6	15 Sep 2010	Fixed some typos, added page numbers, API border cases (Null pointers, empty arrays)
V0.7	15 Sep 2010	Additional Exception: IncompatibleTimeBase