Microsoft HLS Client SDK Application Developer's Guide

V1.2 for Windows 8.1 & Windows Phone 8.1

Table of Contents

1. Purpose

The Microsoft HLS SDK (referred to as the SDK henceforth) is a set of components to assist you in building Windows Store applications that can play back streaming video delivered over the HTTP Live Streaming (HLS) protocol. This document outlines the steps required to use the SDK in your apps. It also provides guidance on the API that allows control over various aspects of the playback experience.

The primary component in the SDK is a custom Media Extension(referred to as the extension henceforth) that adds HLS playback capability to the media pipeline of your app. You can find more on Media Extensions at http://msdn.microsoft.com/en-us/library/windows/apps/hh700365.aspx.

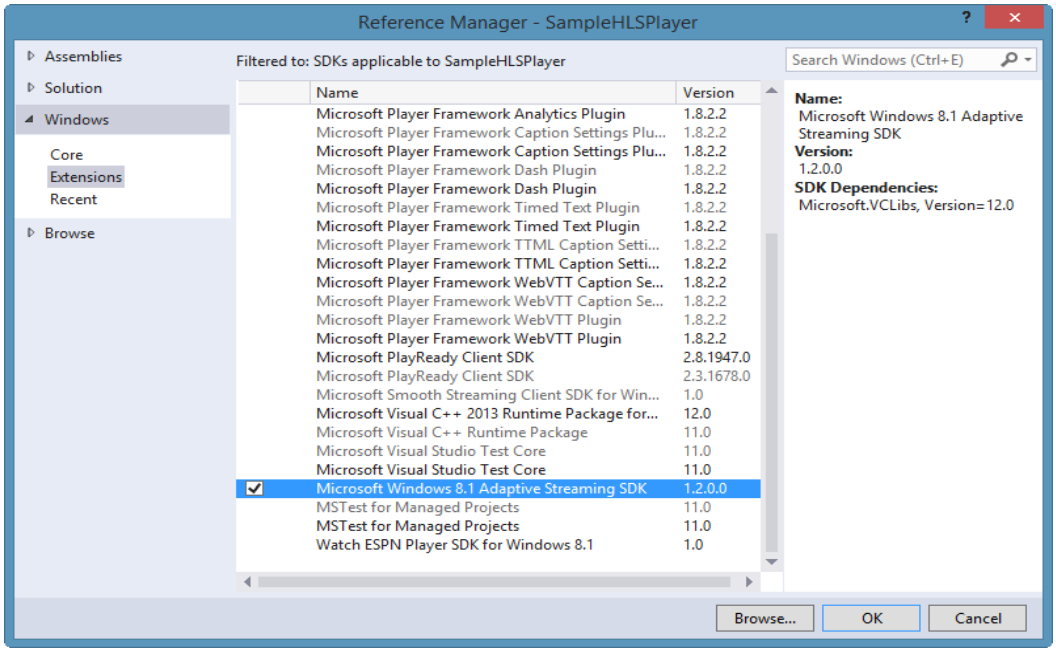
Note : Please note that v1.2 of the HLS SDK is compatible with Windows 8.1 & Windows Phone 8.1 targeted applications only. If you are trying to target Windows 8.0 applications as well, please continue to use v 1.1 of the SDK.

2. Installing the SDK

The SDK is delivered as an Extension SDK for Microsoft Visual Studio. To install it, run the **Microsoft.HLSClient.Setup.vsix** package from the command line.

3. Referencing the SDK from your application

Once you have installed the SDK, to refer to the SDK from your application, open the Add Reference dialog and select Windows Extensions. You should see Microsoft HLS Client SDK listed as shown below.



Select the SDK and click OK to add a reference to your application.

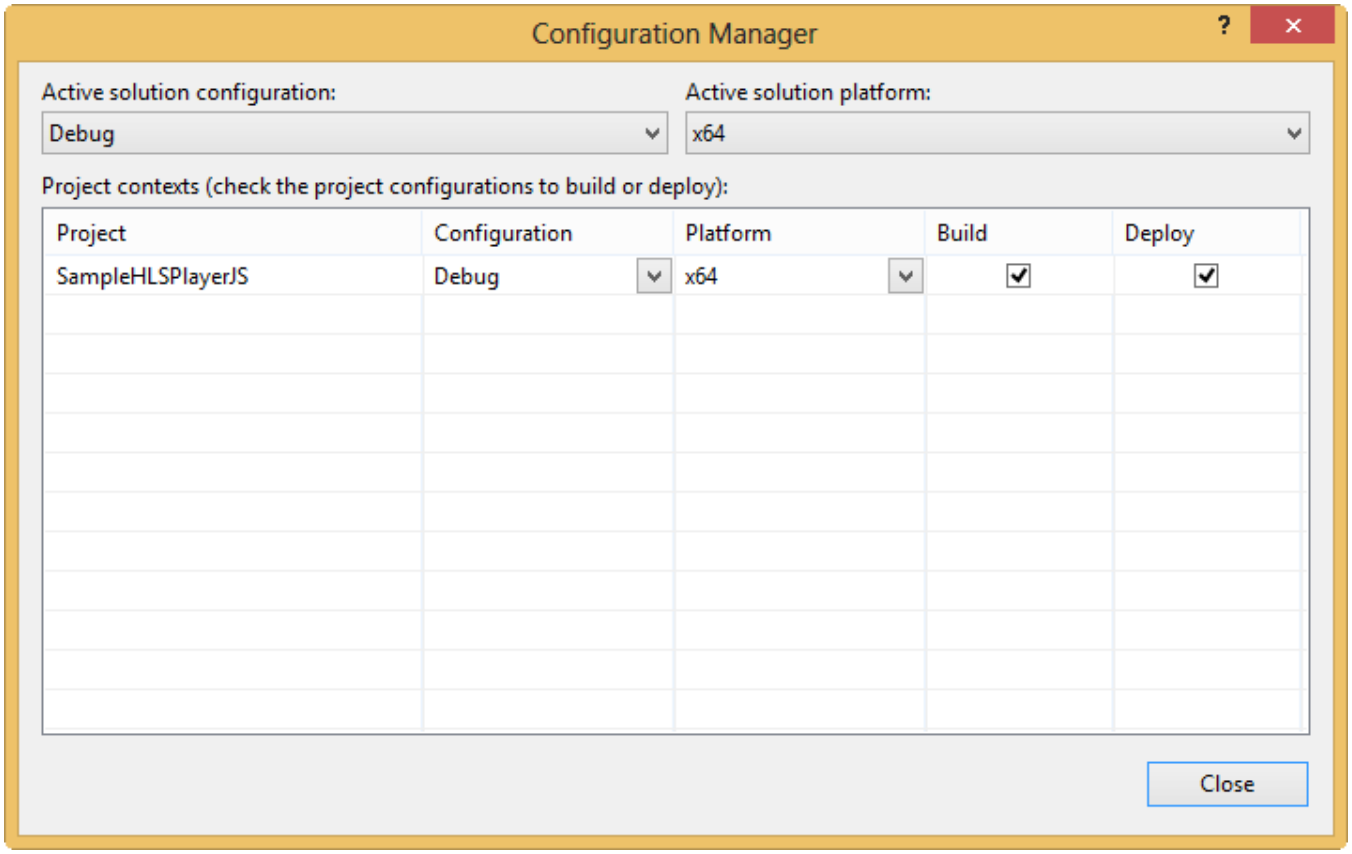
4. Modifying the app manifest

To allow the application to activate and use the custom extension, you will need to manually add details about the extension to your application manifest. From the context menu for your application manifest file (ending with .appxmanifest) in Visual Studio, select "View Code" to open the manifest file, copy and paste the XML (as is) below into your manifest file at the very end, right before the closing **</*Package*>** tag, and save the manifest file.

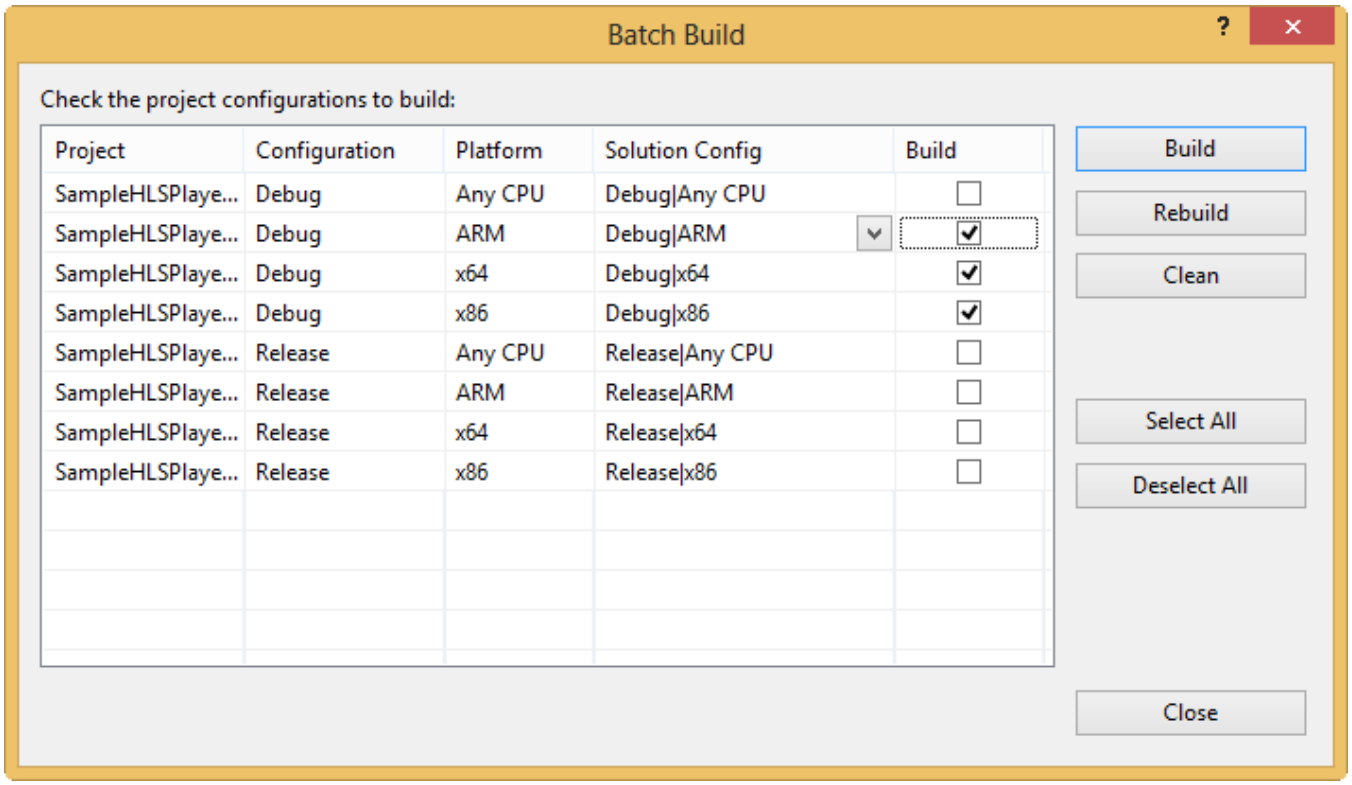
```
<Extensions>
  <Extension Category="windows.activatableClass.inProcessServer">
    <InProcessServer>
      <Path>Microsoft.HLSClient.dll</Path>
      <ActivatableClass ActivatableClassId="Microsoft.HLSClient.HLSPlaylistHandler" ThreadingModel="both" />
    </InProcessServer>
  </Extension>
</Extensions>
```

5. Selecting Processor Architecture

Note that the SDK has a dependency on the Microsoft Visual C++ runtime, which requires that you build your application separately for each processor architecture that you want to target (ARM, x86 and x64). Choosing "Any CPU" as your target processor architecture will not work. To select the processor architecture, use the Visual Studio Configuration manager from the context menu of your solution and change the solution platform to the processor architecture you are targeting.



To build for multiple processor architectures, you can use the Batch Build functionality in Visual Studio.



6. Registering the Extension

Once you have added a reference to the SDK and modified your manifest as outlined above, you will need to add some code to register the extension at runtime. To register a custom extension, you can use the ***Windows.Media.MediaExtensionManager*** type. The extension used for HLS playback is a byte stream handler, for which you can use the ***RegisterByteStreamHandler()*** method on the ***MediaExtensionManager***. The code below shows an example in C#.

```
Windows.Media.MediaExtensionManager extMgr = new MediaExtensionManager();
PropertySet ps = new PropertySet();
ps.Add("ControllerFactory", controllerFactory);
extMgr.RegisterByteStreamHandler("Microsoft.HLSClient.HLSPlaylistHandler", ".m3u8", "application/vnd.apple.mpegurl
", ps);
```

7. Basic Playback

Once you have registered the extension, implementing basic HLS stream playback follows the standard paradigm of using a ***MediaElement*** (XAML) or the ***<Video>*** tag (HTML5) to play back video in a Windows Store app.

For a .Net application, add the ***MediaElement*** to your XAML, and then set the ***Source*** property to the HLS playlist URL – either in code or in the XAML markup. The snippet below shows an example of doing this in the markup. With the AutoPlay property set to true, this content will start playing immediately.

```
<MediaElement x:Name="mediaElem" AutoPlay="True" Source="https://somesite.com/somehlscontent.m3u8"/>
```

Below is another example of doing the same in application code (in C#).

```
public sealed partial class PlayerPage : Page
{
  MediaExtensionManager mediaExtMgr;

  public PlayerPage()
  {
    this.InitializeComponent();

    mediaExtMgr = new MediaExtensionManager();


      PropertySet ps = new PropertySet();
      ps.Add("ControllerFactory", controllerFactory);
      mediaExtMgr.RegisterByteStreamHandler("Microsoft.HLSClient.HLSPlaylistHandler", ".m3u8", "application/vnd.ap
ple.mpegurl", ps);

  }
  protected override void OnNavigatedTo(NavigationEventArgs e)
  {
    mediaElem.MediaOpened += mediaElem_MediaOpened;
    mediaElem.Source = new Uri("https://somesite.com/somehlscontent.m3u8");
  }

  void mediaElem_MediaOpened(object sender, Windows.UI.Xaml.RoutedEventArgs e)
  {
    mediaElem.Play();
  }
}
```

7.1. Basic Playback Control

Playback control functions such as Pause, Stop, Play, Seek etc. all work as expected through the standard ***MediaElement*** WinRT API or the ***<Video>*** tag DOM API's. No special handling is required. As an example, if you are authoring your application using the XAML ***MediaElement***, you can use ***MediaElement.Stop()*** or ***MediaElement.Pause()*** to stop or pause the playback, and can set ***MediaElement.Position*** to seek either forward or backwards into the content.

This document will not discuss the standard ***MediaElement*** or ***<Video>*** tag API's and you are encouraged to look at the appropriate documentation at http://dev.windows.com for further details.

8. Events and Threading

The HLS SDK public API, discussed in the rest of the document, exposes several events that you can handle in your application code. It is important to note that all these events are raised in a background thread. So if you are authoring a XAML application in any of the .Net languages or in C++/CX, and have the need to access and manipulate UI elements in these event handlers in your code, you need to use a mechanism like the ***CoreDispatcher*** to switch execution context to the UI thread. If you are using JavaScript – this is automatically taken care of for you.

9. Controlling HLS playback

The rest of this document focuses on the API that allows you control over specific aspects of HLS playback. All the HLS specific API is exposed through a root component called the HLS controller. To obtain the HLS controller, you will need to create an instance of the ***Microsoft.HLSClient.HLSControllerFactory*** type, and supply that as a named property value to the ***MediaExtensionManager*** while registering the byte stream handler.

The ***HLSControllerFactory.HLSControllerReady*** event is raised by the extension when the controller object is created and ready for use. The code below shows an example of obtaining the controller.

```
MediaExtensionManager mediaExtMgr;
HLSControllerFactory controllerFactory;
IHLSController controller;

public PlayerPage()
{
  this.InitializeComponent();

  controllerFactory = new Microsoft.HLSClient.HLSControllerFactory();
      controllerFactory.HLSControllerReady += controllerFactory_HLSControllerReady;

      mediaExtMgr = new MediaExtensionManager();
      PropertySet ps = new PropertySet();
      ps.Add("ControllerFactory", controllerFactory);
      mediaExtMgr.RegisterByteStreamHandler("Microsoft.HLSClient.HLSPlaylistHandler", ".m3u8", "application/vnd.ap
ple.mpegurl", ps);

}

void controllerFactory_HLSControllerReady (IHLSControllerFactory sender, IHLSController args)
{
  controller = args;
}
```

Note that the property key used in adding the factory to the property set needs to be the exact string literal "***ControllerFactory***" as shown above in the code snippet.

The ***IHLSController*** interface exposes properties that lets you control certain aspects of the playback experience. The table below lists the properties. We will discuss these properties and their usage in later sections where appropriate.

Table 1. IHLSController properties

Name	Type	Access	Purpose
ID	String	Read Only	Controller Identifier.
IsValid	boolean	Read Only	Indicates whether the controller is attached to a valid playback session any more or not. You should not attempt to use a controller instance that reports the IsValid property as false.

MinimumBufferLength	TimeSpan	Read/Write	Specify the minimum length of the read ahead buffer maintained by the extension. Default value is 5 times the target segment duration as specified in the HLS playlist.
BitrateChangeNotificationInterval	TimeSpan	Read/Write	The interval at which the bitrate monitor notifies the extension of any required bitrate changes. Default value is 15 seconds.
EnableAdaptiveBitrateMonitor	boolean	Read/Write	Turns the bitrate monitor on(true) or off(false). Default value is true.
Playlist	IHLSPlaylist	Read Only	Provides access to the HLS playlist information.
SegmentTryLimitOnBitrateSwitch	unsigned int	Read/Write	Determines the maximum playback time in number of segments of playback, over which the runtime attempts to complete a bitrate switch before giving up (default = 3)
AllowSegmentSkipOnSegmentFailure	bool	Read/Write	Determines if the runtime is allowed to skip a segment that fails to download (default = true)
ForceKeyFrameMatchOnSeek	bool	Read/Write	Determines whether the runtime starts playback at a key frame following a seek operation (default = true)
AutoAdjustScrubbingBitrate	bool	Read/Write	Determines whether bitrates are automatically shifted down to enable scrubbing (default = false)
AutoAdjustTrickPlayBitrate	bool	Read/Write	Determines whether bitrates are automatically shifted down to enable faster than normal rate of playback (default = true)
UseTimeAveragedNetworkMeasure	bool	Read/Write	If set to false, bitrate monitoring results is averaged over the time interval specified in BitrateChangeNotificationInterval. Otherwise a bitrate switch is suggested based on single segment download performances and are more "instantaneous". The default is true.
MatchSegmentsUsing	SegmentMatchCriterion	Read/Write	Determines whether to use Sequence Number or Program Date Time based segment matching during a bitrate switch
MaximumToleranceForBitrateDownshift	Float	Read/Write	The maximum amount of tolerance allowed by the runtime before a bitrate downshift occurs. Expressed as a multiplier of the currently playing bitrate i.e. if the current bitrate is 500 kbps, a tolerance value of 0. 2 will cause the runtime to not shift down

			until the measured download speed falls below 490 kbps. The default is 0.
MinimumPaddingForBitrateUpshift	Float	Read/Write	The minimum amount of padding required by the runtime before a bitrate upshift occurs. Expressed as a multiplier of the target bitrate i.e. if the current bitrate is 500 kbps and the next higher bitrate is 1000 kbps, with a padding of 0.5 the download bandwidth has to be at least 1500 kbps before the runtime will shift up. The default is 0.35.
MinimumLiveLatency	TimeSpan	Read/Write	The minimum distance from the tail of a live playlist that live playback should start at. The default is 3 times the average segment duration.
PrefetchDuration	TimeSpan	Read/Write	The minimum length of content that needs to be downloaded before playback can start. The default is 0.
TrackTypeFilter	TrackType	Read/Write	Can be used to force the runtime to play audio or video only. The default is TrackType.BOTH, and the allowed values are BOTH, AUDIO and VIDEO. Can only be set in the ControllerReady event handler.
UpshiftBitrateInSteps	Boolean	Read/Write	Forces the runtime to shift up by stepping through all bitrates in between instead of going directly to the target bitrate. The default is false.

9.1. Buffering Control

The runtime maintains a read ahead buffer in an attempt to avoid buffering during playback. In other words as a segment is being played back, subsequent segments are downloaded, parsed and readied for playback in the background and this process continues until a certain amount of content is in memory.

The ***IHLSController.MinimumBufferLength*** property, expressed as a TimeSpan, can be used to control the buffering behavior. Buffering starts at the beginning of playback for the initial content load and then resumes again whenever the buffer length (i.e. the duration of content in memory between the current playback position and the end of the buffer) falls below ***MinimumBufferLength***. Buffering stops whenever the buffer length exceeds the ***MinimumBufferLength*** value or the stream ends.

You should carefully consider the impact of changing the ***MinimumBufferLength*** property. To ensure smooth playback the extension checks the buffer length only when playback switches from one segment to the next one. If this value is set low enough you may see buffering (i.e. playback halted while data gets downloaded) when these segment switches occur.

The default value of the ***MinimumBufferLength*** property is set to four times the ***TargetDuration*** property of the playlist i.e. roughly equal to four times the length of a single segment of content.

9.2. Prefetching

At times just ensuring a long enough read buffer is maintained is not enough. The runtime attempts to start playback as soon as the first segment is read and utilizes the playback time to prepare the buffer by downloading subsequent segments. However in certain scenarios, especially when the segment length is very small and the bitrate played is relatively high, the playback duration of a segment may not be enough time for the runtime to download and prepare the very next segment – which then causes the system to buffer again.

To avoid this you can specify that a certain duration of content be pre-fetched and processed before playback starts. The ***IHLSConroller.PrefetchDuration*** property on the controller allows you to stipulate this value as a TimeSpan. By default, the runtime sets the ***PrefetchDuration*** to five seconds whenever the target segment duration is less than five seconds.

9.3. Adaptive Bitrate Monitoring

The extension actively monitors network performance by measuring download speeds as various assets get downloaded during playback. As and when bandwidth fluctuates, the extension can automatically switch the currently playing bitrate to one that is closer to the currently effective bandwidth.

Note that this section refers to the ***Playlist*** object. The ***Playlist*** object is discussed in more details in section 9.

The bitrate switch is a two phase process. The bandwidth monitor can suggest a bitrate switch at any point in time during playback. When this suggestion is received by the extension, it schedules a switch and raises a ***BitrateSwitchSuggested*** event on the ***Playlist*** object. The actual switch however is completed the next time a segment completes and playback moves over to the next segment. At this time, upon completion of the bitrate switch, the extension raises the ***BitrateSwitchCompleted*** event.

The ***IHLSBitrateSwitchEventArgs*** interface supplies more details about the bitrate switch through the ***FromBitrate*** and ***ToBitrate*** properties , respectively providing the values of the bitrate that the stream was playing at and the one that it will switch to.

The code snippet below shows an example of handling these events.

```
void mediaElem_MediaOpened(object sender, Windows.UI.Xaml.RoutedEventArgs e)
{
  controller.Playlist.BitrateSwitchSuggested += Playlist_BitrateSwitchSuggested;
  controller.Playlist.BitrateSwitchCompleted += Playlist_BitrateSwitchCompleted;
  controller.Playlist.BitrateSwitchCancelled += Playlist_BitrateSwitchCancelled;
  mediaElem.Play();
}
```

```
void Playlist_BitrateSwitchSuggested(IHLSPlaylist sender, IHLSBitrateSwitchEventArgs args)
{
  //do something on bitrate switch suggestion

  var switchingFrom = args.FromBitrate;
  var switchingTo = args.ToBitrate;

  if (weNeedToCancel)
    args.Cancel = true;
}

void Playlist_BitrateSwitchCancelled(IHLSPlaylist sender, IHLSBitrateSwitchEventArgs args)
{
  //do something on bitrate switch cancellation
}

void Playlist_BitrateSwitchCompleted(IHLSPlaylist sender, IHLSBitrateSwitchEventArgs args)
{
  //do something on bitrate switch completion
}
```

Also note that that while the ***BitrateSwitchSuggested*** event is raised only once, the ***BitrateSwitchCompleted*** event can be raised more than once – once for each track type (AUDIO and VIDEO) that switches over to the new bitrate. The ***IHLSBitrateSwitchEventArgs.ForTrackType*** property can be queried to find out which track switch is being completed.

9.3.1. Turning bitrate monitoring off

The ***EnableAdaptiveBitrateMonitor*** property on the controller can be set to false to turn automatic bitrate switching off during playback. Setting the property to true turns switching back on again. You should carefully consider the implications of turning bitrate monitoring off – not having the ability to switch bitrates in the face of diminishing network bandwidth can cause a degraded playback experience due to buffering, as well as can deprive the user of a higher quality playback experience when the network bandwidth increases but the extension does not switch bitrates up because monitoring is turned off.

9.3.2. Instant notification vs. Time Averaging

There are two modes in which you can use the network monitor to monitor for network changes. You can use the ***IHLSController.UseTimeAveragedNetworkMeasure*** property to switch between the modes. When this property is set to false (this is the default), the network monitor suggests a bitrate switch as soon a single segment download reports a change in bitrate from the currently playing bitrate. This mode is used to provide an "instant" notification where by content can be switched between bitrates more aggressively.

However there can be times when the network may fluctuate suddenly for a very small amount of time between otherwise steady behavior, and not using a time averaged mechanism can lead to false positives and unnecessary bitrate switches.

Alternatively setting this property to true causes the bitrate monitor to notify the extension that a bitrate change is due, only at a specified interval. Every time this interval elapses, the monitor averages network speed data that it has collected since the previous occurrence of this interval, compares it to the bitrate that is currently playing back, and suggests a bitrate change if the average differs from the currently playing bitrate.

Note that using time averaging only affects bitrate upshift decisions. When a fall in bandwidth requires a down shift, the runtime issues the downshift disregarding any time averaging settings. Also note that in either case the bitrate suggested is the closest lower variant bitrate that matches the computed average.

If you are using time averaged measures, you can change the value of the notification interval by setting the ***BitrateChangeNotificationInterval*** property on the controller to the desired time value measured in milliseconds. The default value of this interval is set to 5000 milliseconds. Setting this value too high or too low can cause unnecessary buffering. A generally good value for this is about the average length of a segment duration.

9.3.3. Step Shifting

Step shifting allows you to specify that a shift from one bitrate to another should always happen in steps i.e. the runtime moves through all the bitrates in between before it lands on the target bitrate. This is helpful when the bitrate is shifting upwards. The default behavior is to try to shift to the highest possible bitrate allowed by the currently available bandwidth and other settings such as any applied padding. However shifts are non-deterministic and dependent and on occasions where the bandwidth fluctuates frequently such an upshift may keep failing keeping the viewer at the current bitrate. In situations like that it may be easier to shift the viewer up through bitrates where by even if the highest allowed bitrate is not attainable for some reason, the player can provide the viewer a relatively better experience by shifting to a bitrate in between thus maximizing the quality.

The HLS runtime supports step shifting for upshifts only. To set this you can set the ***UpshiftBitrateInSteps*** property on the controller to true. The default value for this property is false when running on Windows 8.1 but true when running on Windows Phone 8.1.

9.3.4. Cancelling a bitrate switch

The ***IHLSBitrateSwitchEventArgs.Cancel*** property also allows you to programmatically cancel a suggested bitrate switch. When a bitrate switch is cancelled the extension raises the ***BitrateSwitchCancelled*** event on the current playlist. Note that setting the ***Cancel*** property in handling the ***BitrateSwitchCancelled*** or the ***BitrateSwitchCompleted*** events has no effect.

9.3.5. Setting bitrate bounds

HLS variant playlists define the list of bitrates that are supported by that stream. However, the extension allows you to stipulate a minimum and a maximum bitrate value such that playback is never

switched beyond those values. You can use the ***MaximumAllowedBitrate*** and ***MinimumAllowedBitrate*** properties on the Playlist object to specify these values measured in bits per second. So to set the ***MaximumAllowedBitrate*** to say 1.5 mbps, you would write code like this:

```
controller.Playlist.MaximumAllowedBitrate = (uint)(1024 * 1024 * 1.5);
```

Unless these properties are changed in code, they provide the maximum and minimum bitrate values as specified in the master playlist. Once you change these values, if you need to get the original maximum and minimum bounds, you can use the ***GetVariantStreams()*** method on the ***Playlist*** object to get the array of the variants and sort the ***Bitrate*** property values on the returned variants to obtain the original maximum and minimum values.

Note that setting these properties to values that are beyond the allowed minimum and maximum bitrates as specified in the HLS playlist will throw exceptions.

9.3.6. Setting start bitrate

The extension starts initial playback at the first bitrate it encounters in the list of bitrates as specified in the variant playlist. However you can programmatically specify the start bitrate by setting the value of the ***StartBitrate*** property on the ***Playlist*** object, measured in bits per second.

Note that setting the start bitrate property once the media has started playing has no effect. To effectively apply a start bitrate value you should set the ***StartBitrate*** property in your code before the media starts playing back. A good place to do this is the ***HLSControllerReady*** event handler.

9.3.7. Handling Audio Only variants

HLS content often has its lowest bitrate encoded to be an audio only stream. In scenarios like that, if you are building a player experience you may want to know when a bitrate switch causes playback to move from an audio/video to an audio only experience or vice versa, as you might need to adjust your UI accordingly . In cases of such a switch, the extension raises the ***StreamSelectionChanged*** event on the playlist, right after the ***BitrateSwitchCompleted*** event. The ***IHLSStreamSelectionChangedEventArgs*** type exposes two properties named ***From*** and ***To***, both of the enumeration type ***TrackType*** with possible values being – ***AUDIO***, ***VIDEO*** or ***BOTH***. The code snippet below shows an example of handling this event to change the UI.

```
…
//handle the stream selection changed event
controller.Playlist.StreamSelectionChanged += Playlist_StreamSelectionChanged;
…


void Playlist_StreamSelectionChanged(IHLSPlaylist sender,IHLSStreamSelectionChangedEventArgs args)
{
  Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
  {
    if (args.To == TrackType.AUDIO)
    {
```

```
      //show a message to let user know that we are playing audio only
      gridAudioOnly.Opacity = 1.0;
    }
    else if (args.To == TrackType.BOTH)
    {
      //hide audio only message
      gridAudioOnly.Opacity = 0.0;
    }
  });
}
```

9.3.8. Adjusting bitrate tolerance limits

The default bitrate switching logic in the SDK switches bitrates aggressively as soon as the current monitored value falls below or moves above the currently playing bitrate by even 1 bit per second. However in certain cases you may want to provide a degree of tolerance whereby if the monitored bitrate hovers close to the bitrate being played you want to avoid a bitrate switch. This helps in ironing out situations where the fluctuation was momentary and a switch was really not needed.

You can do so by specifying a percentage value in the ***MinimumPaddingForBitrateUpshift*** and the ***MaximumToleranceForBitrateDownshift*** properties on the ***Controller*** object. Both these properties represent percentages expressed as fractional values. As an example let's assume that the playback is currently at 900 kbps and the next higher bitrate available is 1200 kbps, while the next lower bitrate available is 600 kbps. If you specify the value 0.5 for the ***MinimumPaddingForBitrateUpshift***, the runtime would ensure that an upshift does not happen until the download speed measures at least 1800 kbps i.e. 50% on top of what the next higher bitrate is. Similarly if you specify a value of 0.1 for the ***MaximumToleranceForBitrateDownshift*** property, a down shift will not happen unless the recorded download speed falls below 810 kbps i.e. 10% less than the currently playing bitrate. This can help avoid unnecessary switches in situations where little network fluctuations are common. The default value for ***MinimumPaddingForBitrateUpshift*** is 0.35 while that for ***MaximumToleranceForBitrateDownshift*** is 0.

9.3.9. Limiting bitrate switch execution time

As we mentioned before, bitrate switching is a two-step process – the first step notifies the runtime that a switch is required. The runtime then attempts to determine the appropriate segment to switch over at, attempts to download, and as the second step switches over when appropriate. The time required to download and match content that happens in between these two steps can take a while and can sometimes have a negative impact on playback as the runtime tries to download segments from multiple bitrates (the one that is playing and the one you are trying to switch to) at the same time. To minimize this you can set the ***IHLSController.SegmentTryLimitOnBitrateSwitch*** property to a positive value. This property value determines the number of segments worth of playback time that the runtime will try to execute a suggested bitrate switch before it gives up (and waits for the next notification). The default value is 2.

9.3.10. Obtaining the download speed

The HLS runtime constantly measures download speed. This measured speed is what is used to make bitrate switching decisions. If for reasons specific to your application, you do need to know what the currently measured download speed is – you can use the ***IHLSController.GetLastMeasuredBandwidth()*** method. The value returned is the last measured download speed in bits per second. Note that this value is not indicative of the overall network bandwidth available to the device – this is only a measure of the download speed as measured by the HLS runtime. Also note that this method only returns a valid value if the bitrate monitoring is turned on and the playlist being played is that of a multi-bitrate type.

9.4. Seeking

Seeking using the HLS runtime does not require any special API. You can affect seeking just by updating the ***MediaElement.Position*** property (or the equivalent property in the <video> tag) to the position you would want to seek, and the runtime takes care of the rest. There are however a few things to keep in mind as discussed below.

9.4.1. Note about Seeking and Slider controls

In most player implementations, some sort of a UI control is used that allows viewers to seek by simply dragging or sliding an indicator to the desired position. A common type of control used for a situation like this is a slider control.

A common mistake to make in implementations is to handle every incremental change in the slider control's current thumb position and use that to update the Position property of the MediaElement (or equivalent for the <video> tag). Note that every time that is done, the runtime tries to effect a seek to that position – all the while with the viewer not having completed his dragging the thumb to the desired position – resulting in a very large number of individual seek actions on the way to the final desired position. A seek is an expensive operation, especially in case of streamed content where it involves actually downloading the content (that portion of the content may not be in memory yet) before you can seek to it. An implementation like the one described above can easily flood the runtime with hundreds of HTTP calls before it the user reaches the desired spot.

A better way to handle this is to override the drag behavior of the slider control (or similar) and not update the ***MediaElement.Position*** property (or equivalent <video> tag property) while the thumb is being dragged, but rather update it with the final position, only once the viewer has stopped dragging the thumb.

9.5. Seeking and Live Streams

HLS typically implements live streaming using a "sliding window" mechanism, where segments from the beginning of the live playlist are dropped from the server and the playlist over time, while new segments get added to the end of the playlist. HLS also supports a special kind of live playlist called the

Event playlist (indicated by the value of the EXT-X-PLAYLISTTYPE tag being the string 'EVENT'). In Event playlists, no segments are ever dropped from the playlist as playback progresses, with only new segments added to the end of the playlist over time.

The HLS SDK exposes the ***IHLSSlidingWindow*** type to provide running information about the sliding window during live playback. You can access sliding window information through the ***IHLSPlaylist.SlidingWindow*** property at any time. You can also subscribe to the ***IHLSPlaylist.SlidingWindowChanged*** event if you want to be notified whenever the sliding window information changes.

The ***IHLSSlidingWindow*** exposes the beginning and the end of the current sliding window using the ***StartTimestamp*** and ***EndTimestamp*** properties. The current live playback position always falls between the start and the end timestamps, and is exposed through the ***LivePosition*** property. Note that the ***IHLSSlidingWindow.StartTimestamp*** never changes for an event playlist – with only the tail of the sliding window moving forward in time along with the live playback position.

The HLS runtime supports seeking backwards in a live playlist (this functionality is sometimes also called Live DVR). You can set the position to any point between the start of the sliding window and the current live position and playback resumes at that position and continues. An attempt to set position to any point at or beyond the current live position automatically causes playback to resume at the current live position – this is also the mechanism that you can use to put playback "back to live" from your application code. The code sample below shows a sample of setting playback back to the live position – we deliberately add half a second to the current live position to ensure that the playback snaps to live.

```
if (_controller != null && _controller.IsValid
  && _controller.Playlist != null
  && _controller.Playlist.IsLive)
{
  _player.Position = _controller.Playlist.SlidingWindow.LivePosition + TimeSpan.FromMilliseconds(500);
}
```

To create a scrubber that reflects live playback such that a viewer can seek back into live content, you can use a regular slider control and bind its minimum and maximum bounds to the ***StartTimeStamp*** and ***EndTimeStamp*** values and the current position to the ***LivePosition*** value. As playback continues you will need to do this repeatedly as the sliding window moves – doing this using a timer event or in the ***SlidingWindowChanged*** event handler are both acceptable ways. If you are using the ***PlayerFramework***, you can set the ***MediaPlayer.StartTime***, ***MediaPlayer.EndTime*** and the ***MediaPlayer.LivePosition*** properties to get the same effect.

9.6. Resuming a Paused Live Stream

If the viewer pauses a live stream, the default behavior of the HLS runtime on resumption of the paused stream is to start the stream from the current live position i.e. the viewer loses the portion of content that played in between the time the stream was paused and later resumed. This behavior can

be modified by setting the ***ResumeLiveFromPausedOrEarliest*** property to true on the ***IHLSController*** (the default is false). When set to true, this attempts to start the live playlist from the paused position. If segments are dropped between the pause and resumption such that the paused position is no longer available, playback starts at the earliest point possible i.e. the beginning of the live playlist at the time of resumption.

9.7. Variable Rate Playback

Variable Rate playback allows the viewer to play the same content at more or less than normal speeds in either forward or reverse direction. One of the most common uses of variable rate playback is to implement "Fast Forward" or "Rewind" functionality. The HLS runtime supports the following variable playback rates:

Rate	Purpose
1.0	Normal speed playback in forward direction
0.0	Frame by frame playback – discussed more in the next section (scrubbing)
0.5	Playback at half the normal speed in forward direction
2.0	Playback at 2 times the normal speed in forward direction
4.0	Playback at 4 times the normal speed in forward direction
8.0	Playback at 8 times the normal speed in forward direction
12.0	Playback at 12 times the normal speed in forward direction
-2.0	Playback at 2 times the normal speed in reverse direction
-4.0	Playback at 4 times the normal speed in reverse direction
-8.0	Playback at 8 times the normal speed in reverse direction
-12.0	Playback at 12 times the normal speed in reverse direction

To change the playback to any of the above rates, simply set the ***MediaElement.PlaybackRate*** property (or equivalent property on the <video> tag) to the appropriate rate value listed in the above table.

9.7.1. Note about Variable Rate playback

Since HLS content is streamed over HTTP, one challenge in implementing "faster than normal" rates of playback is to have the downloader keep up with the playback rate i.e. when playing at higher speeds, playback may reach a point in the stream where the respective content has not yet been downloaded, and stall until the runtime downloader catches up.

To avoid this scenario, in cases where the multi–bitrate content is available, the runtime attempts to shift the bitrate of the content down to lower values to assist in faster downloads to keep up with higher playback rates. The extent to which the bitrate may get down-shifted depends on several factors – the playback rate, current network bandwidth, availability of bitrates lower than the one currently playing etc. The result of such a downshift is that the viewer will see playback at lower bitrate content as long as the rate of playback is higher than normal, but once playback resumes at normal rate, the

standard adaptive bitrate switching will shift the bitrate back up to the appropriate value at the next opportunity.

Also note that in case the source content is only available at a single bitrate, this downshift is not possible. While the runtime does not explicitly turn off variable rate playback for such streams, you should be careful enabling playback rate change in your player for such scenarios as variable rate playback at rates higher than normal may not yield the desired experience.

Lastly note that the ***IHLSController.AutoAdjustTrickPlayBitrate*** property can be set to false (default is true) to turn this bitrate downshifting behavior off.

9.8. Scrubbing

Like seeking, scrubbing allows the viewer to reposition the playback to a specific point in the stream in either direction. However, unlike seeking, scrubbing allows display of individual frames as the viewer navigates to the desired point in the content using a slider control or something similar.

To enable scrubbing, the playback rate of the MediaElement (or the <video> tag) needs to be set to 0.0. The rest of the implementation is similar to seeking, in that navigating to points in the stream is achieved simply by setting Position property to the desired value. However, unlike seeking where setting the Position property continuously while a slider thumb is dragged around is discouraged, scrubbing does not have that limitation.

Note that just like other variable rates, scrubbing too can attempt to shift the bitrate down, in order for the downloader to keep up with the pace at which a viewer might move the slider thumb. As such, as discussed in the previous section, it is advised to enable scrubbing only with multi-bitrate content and not with content that is only available at a single bitrate. However bitrate downshifting is not the default behavior for scrubbing, and you can set the ***IHLSController.AutoAdjustScrubbingBitrate*** property to true to enable this behavior – the default is set to false. Our advice is to set this property to true right before the viewer starts scrubbing and set it to false immediately after.

9.9. Scrubbing, Variable Rate Playback & Live streams

Note that in the current version of the runtime, scrubbing and variable rate playback features are only available for "non-live" streams.

10. Playlist Information

The ***IHLSPlaylist*** describes an HLS playlist. The ***IHLSController.Playlist*** property provides you access to information about the currently playing HLS playlist.

Table 2. IHLSPlaylist properties

Name	Type	Access	Purpose

Url	string	Read Only	The URL to the playlist that is currently playing.
IsMaster	boolean	Read Only	Specifies if the playlist currently playing is a master playlist containing multiple bitrate variants.
IsLive	boolean	Read Only	Specifies if the playlist being played is a live playlist or not. True if Live, false otherwise. The value in this variable is not reliable until the MediaOpened event in the MediaElement (or equivalent event on the <video> tag) has been raised, and should be used accordingly.
PlaylistType	HLSPlaylistType	Read Only	Allowed values are NONE for master playlists and EVENT , VOD or NONE for media playlists. Please refer to the documentation for the EXT-X-PLAYLIST-TYPE tag in the HLS spec for more details.
ActiveVariantStream	IHLSVariantStream	Read Only	The currently active variant. Not applicable if IsMaster is false.
MaximumAllowedBitrate	unsigned int	Read/Write	Maximum bitrate that the bitrate monitor can switch to. Default value is the highest bitrate specified in the playlist. Not applicable if IsMaster is false.
MinimumAllowedBitrate	unsigned int	Read/Write	Minimum bitrate that the bitrate monitor can switch to. Default value is the lowest bitrate specified in the playlist. Not applicable if IsMaster is false.
StartBitrate	unsigned int	Read/Write	The initial bitrate at which the playlist starts playing. Default value is the first bitrate listed in the playlist. Not applicable if IsMaster is false.
TargetDuration	unsigned int	Read Only	Provides the target segment duration of the segments in a media playlist. Only applicable if IsMaster is false.
BaseSequenceNumber	unsigned int	Read Only	Provides the base sequence number for segment numbering in a media playlist. Only applicable if IsMaster is false.
SegmentCount	unsigned int	Read Only	Provides the total number of segments in the media playlist. Only applicable if IsMaster is false.
ParentStream	IHLSVariantStream	Read Only	The parent stream for the media playlist in a multi bitrate scenario.
PlaylistType	HLSPlaylistType	Read Only	The HLS Playlist type ("VOD" ,"EVENT" or "NONE")

10.1. Variant Playlists

The extension can play both master and media playlists. Master playlists support adaptive bitrate switching by providing a list of variant streams encoded at different bitrates with each stream entry pointing to a media playlist. You can query the ***IsMaster*** property on the Playlist object to determine

whether the playlist is a master playlist or not. None of the properties on the ***Playlist*** object except ***Url*** and ***IsMaster*** are applicable unless the playlist is a master playlist.

10.2. Bitrate Monitoring

The ***playlist*** object exposes properties and events that assist in programmatic control over bitrate monitoring and switching. Please refer to section 9.2 for more details.

11. Variant Stream Information

Each bitrate variant and its associated media playlist, in a master playlist, is represented using the ***IHLSVariantStream*** interface. If the playlist is a master (***IHLSPlaylist.IsMaster*** is true) then the ***GetVariantStreams()*** method on the playlist object can be used to obtain the variant streams. The code snippet below shows an example.

```
if (controller.Playlist.IsMaster)
{
  var variantStreams = controller.Playlist.GetVariantStreams();
  foreach (IHLSVariantStream stm in variantStreams)
  {
    //do something with the stream
  }
}
```

The table below lists the properties on the ***IHLSVariantStream*** interface.

Table 3. IHLSVariantStream properties

Name	Type	Access	Purpose
Url	string	Read Only	The URL to the playlist for this bitrate.
IsActive	boolean	Read/Write	Specifies if this variant is currently active.
HasResolution	boolean	Read Only	Specifies if the master playlist contained resolution information for this variant.
HorizontalResolution	unsigned int	Read Only	Horizontal resolution. Valid only if HasResolution is true.
VerticalResolution	unsigned int	Read Only	Vertical resolution. Valid only if HasResolution is true.
Bitrate	unsigned int	Read Only	The bitrate for this variant.
Playlist	IHLSPlaylist	Read Only	The media playlist for this variant stream

11.1. Locking playback to a bitrate

As you have seen earlier, you can turn off adaptive switching. This will cause playback to continue at the bit rate at which the stream was playing when the switching gets turned off. However if you need to actually select the bit rate at which you would want the stream to play, you can do so as well, by calling the ***LockBitrate()*** method on the ***IHLSVariantStream*** interface for the variant that you want to lock playback on to. Note that once a bitrate is locked, although the bitrate monitor continues to run,

adaptive bitrate switching is internally turned off. To revert back to adaptive switching and unlock the playback, call ***IHLSPlaylist.ResetBitrateLock()*** on the root playlist(***Controller.Playlist***).

From a user experience point of view, you can use this feature to give the user a choice of the bitrates by enumerating the variants in the playlist, and then allow the user choose one to play back, or use automatic switching. You should carefully consider using this feature – locking on to a bitrate regardless of network conditions can cause unnecessary buffering, or a lower resolution playback, when a higher resolution is achievable based on available bandwidth.

11.2. Locking the ActiveVariantStream

The ***IHLSPlaylist.ActiveVariantStream*** provides you access to the currently active variant in case you are playing multi-bitrate content. Whenever a bitrate switch occurs, the active variant changes to the ***IHLSVariantStream*** representing the currently playing bitrate. Since bitrate changes are non-deterministic and can happen anytime, you may need to protect your player side code using the ***ActiveVariantStream*** from having the variant stream change while your code is trying to access it. In order to do that, you can use the ***IHLSController.Lock()***, ***TryLock()*** and ***Unlock()*** methods. The ***Lock()*** method attempts to lock the runtime preventing it from executing a bitrate switch until it is released through a call to ***Unlock()***. The ***Lock()*** method blocks until it can acquire the lock. If you do not want to block execution, you can use ***TryLock()*** which returns true if a lock was successful and false otherwise. In either case a matching call to ***Unlock()*** is mandatory. The sample below shows appropriate usage.

```
if (controller.TryLock())
{
  //do something with the active variant
  controller.Unlock();
}
```

As an alternative, you can also use standard exception handling mechanisms in your language to guard against the same issue.

11.3. Using alternate audio

HLS supports both alternate audio and video renditions. Although this version of the HLS SDK for Windows 8 exposes metadata about both audio and video renditions, it only supports playing back alternate audio renditions, with alternate video support coming in future versions.

To enumerate the audio renditions, you can use the ***IHLSVariantStream.GetAudioRenditions()*** method on the variant to get the collection of available audio renditions.

An alternate rendition is represented by the ***IHLSAlternateRendition*** type. The table below lists the properties for this interface.

Table 4. IHLSAlternateRendition properties

Name	Type	Access	Purpose

Url	string	Read Only	The URL to the playlist for this rendition.
IsActive	boolean	Read/Write	Specifies if this rendition is currently active. Set to true to activate an alternate rendition. Set to deactivate and switch to main track.
IsDefault	boolean	Read Only	Indicates if this is the default choice of rendition. Refer to the EXT-X-MEDIA tag documentation in the HLS specification for more details.
IsForced	boolean	Read Only	Whether this rendition should be forced to playback. Refer to the EXT-X-MEDIA tag documentation in the HLS specification for more details.
IsAutoSelect	boolean	Read Only	Whether this rendition should be automatically selected in the absence of a user preference. Refer to the EXT-X-MEDIA tag documentation in the HLS specification for more details.
Type	string	Read Only	Specifies the type of the alternate rendition. Valid values are AUDIO, VIDEO or SUBTITLE. Refer to the EXT-X-MEDIA tag documentation in the HLS specification for more details.
GroupID	string	Read Only	Identifies the mutually exclusive group this rendition belongs to. Refer to the EXT-X-MEDIA tag documentation in the HLS specification for more details.
Language	string	Read Only	RFC 5646 language tag defining the primary language for the rendition. Refer to the EXT-X-MEDIA tag documentation in the HLS specification for more details.
Name	string	Read Only	Descriptive name of the rendition. Refer to the EXT-X-MEDIA tag documentation in the HLS specification for more details.

11.3.1. Setting Alternate Audio Before Playback Starts

You may have a need to specify a non-default alternate audio rendition to be the one with which playback starts. To do this you will need to specify the alternate audio rendition using the currently active variant. However the HLS runtime sets the active variant past the ***ControlleReady*** event – which means you cannot do this in your ***ControllerReady*** event handler. The appropriate event to do this prior to the start of playback is to use the ***InitialBitrateSelected*** event on the controller which occurs post ***ControllerReady*** but before any playback occurs. Within this handler you will find the ***ActiveVariant*** property on the master playlist to be available, and you can then enumerate and activate an alternate audio rendition to start playback with, as described above.

Note that it is very important to call ***IHLSInitialBitrateSelectedEventArgs.Submit()*** at the end of your work in the handler before you return. This method should be called regardless of any work you do in the handler, or even if you are not really doing any work – as long as you have decided to subscribe to the event and have a handler. Not calling this method from this event handler will cause the runtime to block indefinitely.

11.4. Using subtitles

To enumerate subtitle renditions you can use the Get***SubtitleRenditions()*** method similar to audio renditions. The extension does not attempt to parse or render subtitle content in a subtitle playlist – but expects the player to do so. Consequently, the subtitle related API only exposes the URI to the subtitle resource for a specific time point and nothing more.

A subtitle rendition, in addition to the ***IHLSAlternateRendition*** interface also implements, ***IHLSSubtitleRendition*** interface. The ***RefreshAsync()*** method on this interface can be used to force the extension to load and parse the subtitle playlist. This method returns the count of the subtitle segments in the subtitle playlist. Equipped with that count, you can use the ***GetSubtitleLocatorByIndex()*** method to get information about a subtitle segment at a specified index. You can also use the ***GetSubtitleLocatorByPosition()*** method to get the subtitle segment at a specific time point (defined in system ticks i.e. 100 nanosecond units).

Both of these methods return an object that implements ***IHLSSubtitleLocator***. The ***Location*** property on this object provides you the absolute URL to the subtitle segment resource, while the ***StartPosition*** and the ***Duration*** property respectively provide the time point in the presentation the subtitle segment starts at and the duration through which it should be applied. Finally the Index property provides the segment's index in the subtitle playlist.

Note that the HLS SDK also supports in band 608 closed captioning as well as SMPTE TTML closed captioning, but only via the MMPPF plugins built for the SDK.

12. Segment Information

An HLS media playlist is made up of segments where each segment represents a section of the overall media content, encapsulated either in the MPEG 2 transport stream format or as elementary stream data. As playback continues, the HLS runtime exposes information about the segments that may be useful to the player developer.

A segment is represented using the ***IHLSSegment*** interface. The table below lists the properties exposed by the segment object.

Table 5. IHLSSegment properties

Name	Type	Access	Purpose
ParentPlaylist	IHLSPlaylist	Read Only	The parent playlist object.
ForVariant	unsigned int	Read Only	The bitrate for the variant that the segment belongs to (Applicable only in case of multi bitrate scenarios).
MediaType	TrackType	Read Only	Indicates if the segment has both audio and video data or just one kind (BOTH,AUDIO,VIDEO or UNKNOWN)
Url	string	Read Only	The absolute URL for the segment
SequenceNumber	boolean	Read Only	The sequence number for the segment
LoadState	SegmentLoadState	Read Only	The load state for the segment (LOADED or NOTLOADED)

12.1. Accessing Segments

You have a few different ways of accessing segments.

12.1.1. SegmentDataLoaded event

Segment data is loaded by the runtime ahead of time to maintain a read ahead buffer of content in memory. As soon as a segment is downloaded and its data parsed, the runtime raises the ***IHLSPlaylist.SegmentDataLoaded*** event that provides you access to an ***IHLSSegment*** instance through the event arguments. Note that this event is raised only the root playlist (master or media playlist). To receive this event, you will need to attach a handler to the ***Controller.Playlist*** object, and not to a child playlist (***Playlist*** property exposed by the ***IHLSVariantStream*** in case of multi-bitrate scenarios).

12.1.2. SegmentSwitched event

As playback continues, the runtime switches over to the next segment once a segment is played out. It raises the ***IHLSPlaylist.SegmentSwitched*** event right before the next segment starts playing, and the ***IHLSegmentSwitchedEventArgs.FromSegment*** property provides you access to the segment that just finished playing, and the ***IHLSSegmentSwitchedEventArgs.ToSegment*** provides access to the segment that playback is about to switch to. Note that this event is also raised on the root playlist only – and not on children playlists.

12.1.3. Using the GetSegmentBySequenceNumber() method

You can use the ***GetSegmentBySequenceNumber()*** to supply a sequence number and get access to the corresponding segment in memory. Note that this method can only be called on a media playlist (can be the root playlist in case of non multi-bitrate, non variant playlists). Also note that the segment you obtain using this method may not have been loaded and parsed yet, or may have been scavenged after it has been played back by the time you try to access it. To check and ensure that a segment is currently in loaded state, you can check the ***IHLSSegment.LoadState*** property value.

12.2. Accessing Timed Metadata

The HLS specification allows encoders to embed timed metadata in the form of ID3 metadata tags into the transport stream. The ***IHLSSegment*** interface offers a way for you to extract the timed metadata for player level usage. Once you have access to the segment object (using one of the methods described above), and the segment's load state is ***LOADED***, you can call ***IHLSSegment.GetMetadataStreams()*** which returns a collection of ***IHLSID3MetadataStream*** objects (or null if there is no metadata). Each object in the collection represents a distinct stream of metadata tags in the transport stream. The ***IHLSID3MetadataStream.StreamID*** provides the stream identifier (equal to the PID for the metadata stream inside the transport stream), and a collection of ***IHLSID3MetadataPayload*** objects (or null if there are no payloads). Each ***IHLSID3MetadataPayload*** represents a distinct ID3 metadata payload. The ***IHLSID3MetadataPayload.Timestamp*** provides the timestamp for the metadata tag (in 100 nanosecond ticks). To get the actual payload data, you can call the ***IHLSID3MetadataPayload.GetPayload()*** method which returns the ID3 metadata payload as an array of bytes. The sample below shows an example of extracting the timed metadata from a segment and storing it in a program variable for later use.

```
Dictionary<uint, List<IHLSID3MetadataPayload>> _segmentid3tags = new Dictionary<uint,
List<IHLSID3MetadataPayload>>();
void Playlist_SegmentSwitched(IHLSPlaylist sender, IHLSSegmentSwitchEventArgs args)
{

    var ToSegment = args.ToSegment;
    var metadatastreams = ToSegment.GetMetadataStreams();

    if (metadatastreams != null)
    {
      foreach (var stm in metadatastreams)
      {
        var payloads = stm.GetPayloads();
        if (payloads != null)
        {
          if (_segmentid3tags.ContainsKey(stm.StreamID) == false)
            _segmentid3tags.Add(stm.StreamID, payloads.ToList());
          else
            _segmentid3tags[stm.StreamID] = _segmentid3tags[stm.StreamID].Concat(payloads).ToList();
        }
      }
    }

}
```

12.2.1. Parsing Timed Metadata

Each ID3 tag comprises of a header and multiple frames, with each frame containing a frame header and a frame data payload. The frame header contains a frame identifier which is a four character string indicating the type of the frame. The structure of a frame data payload varies from one frame type to another. The ID3 specification defines several well-known frame types, but organizations can easily define their own custom frame types with respective custom frame data structures.

As such, providing a generic parsing mechanism for all possible ID3 tag instances is near impossible. The HLS runtime does provide some parsing of the ID3 tag down to the frame level. You can call

IHLSID3MetadataPayload.ParseFrames() to get a collection of ***IHLSID3TagFrame*** objects – each instance in the collection representing one frame in the tag. The ***IHLSID3TagFrame.Identifier*** returns the frame identifier, and the ***GetFrameData()*** method provides the frame data as an array of bytes.

12.2.2. Synchronizing Timed Metadata with Playback

While you can extract the timed metadata in a segment when a segment switches to the next, or when it gets loaded, performing any sort of action on the metadata tag during playback will mostly require that you do the necessary action when playback reaches the time indicated by the timestamp on the metadata tag. One way of doing this is to run a timer on the player that starts ticking when playback starts, and use the timer's value to act on the metadata tags. The sample below shows an example (it builds on the previous code snippet when the tags were extracted and kept in a dictionary). It also shows how the frame parsing facility is used to parse ID3 tag frames of type PRIV (private frames).

```
void _eventtimer_Tick(object sender, object e)
{
  var timestamptomatch = meHLS.Position.Ticks;

  if (_segmentid3tags.Count > 0)
  {
    foreach (var stmid in _segmentid3tags.Keys)
    {
      var ID3ToRaise = _segmentid3tags[stmid].Where((pld) => { return pld.Timestamp <= (ulong)timestamptomatch;
}).ToList();
      foreach (var itm in ID3ToRaise)
      {
        var pld = itm.GetPayload();
        var frames = itm.ParseFrames();
        foreach (IHLSID3TagFrame frm in frames)
        {
          var data = frm.GetFrameData();
          if (frm.Identifier == "PRIV")
          {
            var privframeparts = Encoding.UTF8.GetString(data, 0, data.Length).Split(new char[] { '\0'});
            string ownerid = privframeparts[0];
            string privdata = privframeparts[1];
            //do something with the data
          }
        }
        _segmentid3tags[stmid].Remove(itm);
      }
    }
  }

}
```

12.3. Accessing CEA 608/708 closed caption data

The HLS runtime also exposes in stream CEA 608/708 closed captioning data. Similar to the process described in the timed metadata section, once you have access to the loaded segment, you can call ***IHLSSegment.GetInbandCCUnits()*** method that returns a collection of ***IHLSInbandCCPayload*** objects (or null if none found). Each ***IHLSInbandCCPayload*** object in turn exposes the time stamp (in 100 ns ticks) as well as method named ***GetPayload()*** that allows extracting the CC payload as an array of bytes. The runtime makes no attempt to parse or render the CC data – it is left for a player level implementation. However if you are using the Microsoft Media Platform Player framework in

conjunction with the HLS runtime, the MMPPF does provide support for parsing and rendering in-stream CC data and will automatically perform the extraction as described above without you having to explicitly use this API.

12.4. Overriding Program Identifiers

An HLS segment is usually represented as a file formatted using the MPEG2 transport stream container format. The MPEG2-TS format can contain multiple streams packetized and multiplexed, with each stream identified by its own identifier. These identifiers are called Program Identifiers or PID's.

On occasions where there are more than one stream of a specific content type (audio, video or metadata) in the TS container, the runtime uses some default logic to determine which PID's to use. For audio or video content the HLS specification mandates that the stream with the smallest numeric PID value be used to playback content, and this is the default behavior of this runtime as well. For HLS Timed Metadata content the runtime uses whichever PID is marked with a stream type of TS Metadata or in the event that such a mapping does not exist in the TS PMT, it attempts to read metadata from the stream with PID value 189 (noted as TS stream identifier private_stream_identifier_1 in the MPEG2 TS specification). For more on HLS Timed Metadata see section 12.2.

However, in certain circumstances you may want to override this behavior and playback one of the streams that would otherwise be ignored. To do this, you can use the ***IHLSPlaylist.SetPIDFilter()*** method. ***SetPIDFilter***() accepts a single parameter named ***pidfilter*** of type IMap<TrackType,unsigned short>. The type ***TrackType*** can be used to specify the content type you want to influence. Note that ***TrackType.Audio*** , ***TrackType.Video*** and ***TrackType.Metadata*** are the only valid values and other values from the ***TrackType*** enumeration will throw exceptions. The second parameter is used to supply the PID of the stream that you want the runtime to use for that content type. The ***pidfilter*** parameter contains pairs of ***TrackType*** keys and PID numeric values. Note that if you supply a PID that does not exist or does not match the content type supplied, the runtime transparently falls back to the original default logic discussed earlier.

To remove an already applied filter, you can use the ***ResetPIDFilter***() method on the playlist type. Calling ***ResetPIDFilter***() without a parameter removes all existing filters, whereas supplying the ***TrackType*** parameter removes the related filter only.

12.5. Accessing Unprocessed Tags

Often content providers introduce custom tags into HLS media playlist to describe things like advertising cues etc. Since these tags are not a part of the HLS specification, the runtime does not try to parse them or take any actions when these tags are encountered, and the tags remain unresolved. However these tags are exposed as is by the API through the ***IHLSSegment.GetUnprocessedTags()*** method, in case the application code wants to take any actions on these tags. Unresolved tags are divided into three groups: ones that appear before a segment (but after the previous segment) – called

PRE, ones that appear between the #EXTINF tag and the segment URI – called WITHIN, and ones that appear after a segment (but before the next segment) – called POST.

The first parameter to the ***IHLSSegment.GetUnprocessedTags()*** method is of the enum type ***UnprocessedTagPlacement*** (allowable values being PRE,POST and WITHIN) and the return value is an array of strings – with each string representing an unprocessed tag. Note that the POST tags for a segment will also be the PRE tags for the next segment.

13. Resource Request Interception

As the runtime plays back content, it requests various kinds of resources over HTTP. These include playlists, media segments (transport stream files or audio files like raw AAC), and key files for AES-128 encrypted content. You may occasionally encounter scenarios where one or more attributes of these resource requests will need to be modified before a request is sent out by the runtime. The runtime allows you to modify the resource request URL before the request is sent. It also allows you to introduce custom cookies or add/modify HTTP headers for the request before it is sent out.

13.1. Default Handling of Resource URL's

The initial playlist URL is provided by the developer by setting the appropriate property on a MediaElement or the <video> tag. The runtime makes no change to this URL and uses it as is. All subsequent URL's for children resources are used as is, only if they are absolute. If however the runtime finds a relative URL for a child resource – such as a media playlist in a master playlist or a segment in a media playlist, it combines the base URL for the parent resource with the relative URL for the resource to arrive at an absolute URL. For instance if a master playlist was found at http://foo.com/someshow/master.m3u8 and it contained an URL entry for a media playlist like "standarddef/index.m3u8" then the runtime will use http://foo.com/someshow/standarddef/index.m3u8 to obtain the media playlist. Continuing in that vein, if the media playlist had a segment URL like "data/segment_1.ts", its absolute URL will be derived as http://foo.com/someshow/standarddef/data/segment_1.ts.

13.2. Modifying resource URL's

Let's look at some common scenarios where you might need to modify a resource URL before a resource is requested. One such scenario might be the need to change the URL scheme. Imagine a case the initial playlist request is made over http://, but subsequent requests for resources such as segments or key files may need to go over https://. However, if those resource URL's are provided as relative entries in the playlist, the default handling will produce absolute URL's using the parent's scheme i.e. http:// and requests may fail. Another scenario may involve adding or modifying the query string portion of an URL to provide some out-of-band data with the request that the player code might know of but the runtime has no control on.

To intercept resource requests you can handle the ***PrepareResourceRequest*** event on the ***IHLSController*** object. Handling this event will allow you to manipulate all resource requests except the very first one (the URL that you set as the source on the MediaElement or the <video> tag) – the very first request is a special case that we will discuss in a later section.

The code snippet below shows an example of modifying URL schemes using resource request interception. In the sample below we modify the URL's for all key files to use https:// instead of http://.

```
void controllerFactory_HLSControllerReady(IHLSControllerFactory sender, IHLSController args)
{

  args.PrepareResourceRequest += controller_PrepareResourceRequest;
  ...
}


void controller_PrepareResourceRequest(IHLSController sender, IHLSResourceRequestEventArgs args)
{
  if (args.Type == ResourceType.KEY && args.ResourceUrl.StartsWith("http:"))
    args.ResourceUrl = args.ResourceUrl.Replace("http:", "https:");
  args.Submit();
  return;
}
```

The ***IHLSResourceRequestEventArgs.Type*** property is of the enumerated type ***ResourceType***, and provides you the type of resource being requested with possible values being ***PLAYLIST***, ***ALTERNATERENDITIONPLAYLIST***, ***SEGMENT*** and ***KEY***. The URL of the resource being requested is exposed through ***IHLSResourceRequestEventArgs.ResourceUrl*** property, and if you need to modify the URL in any way you can simply set this property in your event handler.

Note that it is very important to call ***IHLSResourceRequestEventArgs.Submit()*** at the end of your work in the handler before you return. This method should be called regardless of any work you do in the handler, or even if you are not really doing any work – as long as you have decided to subscribe to the event and have a handler. Not calling this method from either of these events will cause the runtime to block indefinitely.

13.3. Injecting Cookies and HTTP headers

IHLSResourceRequestEventArgs also exposes two methods – namely ***GetHeaders()*** and ***GetCookies()*** both of which return an array of instances of type ***IHttpHeaderEntry*** (which has two properties – namely ***Key*** and ***Value*** both of type string). For the first method this array provides you with a collection of any existing HTTP headers that the runtime was about to send with the request – with the ***key*** property indicating the name of the header and the ***Value*** property its value. The second method provides you with a similar collection – only this time for any cookies being sent.

Note that both of these arrays are immutable – in that changing them do not have any impact on the state of the resource request and they are provided so that you can inspect what the default request

structure looked like. You can use ***IHLSResourceRequestEventArgs.SetHeader()*** to add a custom HTTP header (or change an existing one as found in the header array). If you need to remove a header, you can use the ***RemoveHeader()*** method. Similarly you can use the ***SetCookie()*** method to add a new cookie to the request, or overwrite a cookie that you have already added, and use the ***RemoveCookie()*** method to remove an already added cookie. Note that you can specify an expiration for the cookie along with is name and value, as well as specify if the cookie is meant for Http requests only, and if the cookie is persistent or meant for the current application session only. The cookie expiration is supplied in 100 nanosecond ticks from midnight, January 1st, 1601 (also known as ***FILETIME***). The snippet below shows an example, where an HTTP header is added, another one is removed and a cookie is added before the resource request is submitted.

```
void controller_PrepareResourceRequest(IHLSControllerFactory sender, IHLSResourceRequestEventArgs args)
{
  var headers = args.GetHeaders();
  args.SetHeader("CustomHdr1", "HdrValue1");
  if (headers.Count((he) => { return he.Key == "CustomHdr2"; }) > 0)
      args.RemoveHeader("CustomHdr2");
  var cookies = args.GetCookies();
  args.SetCookie("Test2", "mshlssdk", (ulong)DateTimeOffset.Parse("11/01/2013 06:30:20 AM").ToFileTime(), false, f
alse);

  args.Submit();
  return;
}
```

13.4. Intercepting the initial resource request

To enable interception of the initial request (most likely the master playlist in case of multi-bitrate content or the media playlist otherwise) in your app, you need to register a custom scheme handler extension in addition to the byte stream handler. So your registration code should look like:

```
extMgr = new MediaExtensionManager();
PropertySet ps = new PropertySet();
ps.Add("MimeType", "application/vnd.apple.mpegurl");
ps.Add("ControllerFactory", controllerFactory);
extMgr.RegisterSchemeHandler("Microsoft.HLSClient.HLSPlaylistHandler", "ms-hls:", ps);
extMgr.RegisterSchemeHandler("Microsoft.HLSClient.HLSPlaylistHandler", "ms-hls-s:", ps);
extMgr.RegisterByteStreamHandler("Microsoft.HLSClient.HLSPlaylistHandler", ".m3u8", "application/vnd.apple.mpegurl
", ps);
```

Note that the scheme handler implementation only recognizes URL's with two custom schemes – the first scheme being ***ms-hls:*** and the second one being ***ms-hls-s:*** (for secure access). So once you register the scheme handler, at runtime, before you set the source on a ***MediaElement*** or a ***<Video>*** object, you will need to take the resource URL for the initial resource and replace the ***http:*** scheme in the URL with ***ms-hls:*** (or ***ms-hls-s:*** if the original scheme is ***https:***). The snippet below provides an example.

```
if (url.StartsWith("http:"))
  url = url.Replace("http:", "ms-hls:");
else if (url.StartsWith("https:"))
  url = url.Replace("https:", "ms-hls-s:");
```

```
meHLS.Source = new Uri(url);
```

If an URL with any other scheme is set as the source, the scheme handler will fail and Windows will continue searching for an appropriate handler for the scheme. Note that this does not require any changes to your HLS infrastructure – this is just a temporary transformation done in your code at runtime, and the scheme handler actually transforms the URL back to its original scheme before making the request anyway. This measure exists to prevent any resource requests that are not intended for HLS to not be intercepted by the HLS runtime, as well as to ensure that your code receives the proper interception events for all resource requests starting with the very first one.

Once the scheme handler is registered, you will need to handle the ***ControllerFactory.PrepareResourceRequest*** event in your code to intercept the initial resource request. Note that this event has the exact same signature as that of the ***IHLSController.PrepareResourceRequest*** event discussed above, and you can perform similar tasks in the handler (like URL modification, cookie injection and HTTP header injection), but only in the context of the initial resource request.

14. Track Type Filters

The ***Controller.TrackTypeFilter*** property allows you to force the runtime to play only audio or only video regardless of the type of content being downloaded. The ***TrackTypeFilter*** property accepts values of the enumerated type ***TrackType*** with the only possible values being AUDIO, VIDEO or BOTH, and default being BOTH. Internally once this value is set the runtime avoids loading the necessary infrastructure required for playing the content type that is filtered out.

This is especially useful for handling the background audio playback scenario on Windows Phone 8.1. When you are playing a stream on the background, there is no need to process any video, and in fact processing video puts undue pressure on the system especially if you are running on lower devices. To facilitate that you have the choice of switching the stream down to an audio only bitrate and turning off bitrate monitoring – but what if the playlist is single bitrate, or is multi-bitrate but does not have an audio only variant ? You then have the option to set the ***TrackTypeFilter*** to AUDIO to force the runtime to ignore any video.

Note that this property can only be set in the ***ControllerReady*** event handler.

15. Handling HTTP Errors

Since all HLS resources are obtained over HTTP (or HTTPS) requests, there is always the possibility of one or more such requests failing. This can result in a vital resource not being obtained in time to allow continued playback. The runtime takes several steps to attempt uninterrupted playback even in the face of such failed calls. The table below outlines failures and related runtime behavior.

Table 6. Error Handling

Resource Failed	Runtime Action
Initial Playlist (Master or Media)	Playback fails
Variant Media Playlist	If there are other bitrates to play, the runtime ignores the bitrate for the failed variant and attempts to continue playback using one of the other available bitrates. Bitrate switching (if turned on) may try again to acquire the failed playlist, and if acquisition succeeds, it gets considered for playback – otherwise the runtime continues to ignore the bitrate.
Media Segment	In a multi bitrate scenario, the runtime attempts to use a matching segment from another available bitrate. The order in which the search is done is as follows: 1. All lower bitrates are searched first in descending order of quality 2. If that fails, or if there are not bitrates lower than the current, then the immediately higher bitrate (if existing and available) is searched as well. 3. If that does not work, and segment shifting is allowed then playback jumps to the next segment. If segment shifting is turned off, then playback fails. In a single bitrate scenario, only step c applies. To allow segment shifting i.e. automatic jumping to the next segment on failures, set ***IHLSController.AllowSegmentSkipOnSegmentFailure*** property to true. The default value is true.
Key File (for encrypted content)	If the download of a key resource fails, playback continues until the encrypted segment is required to be played back. At that point playback is put into buffering, and the key file is requested once more. If it is successfully obtained, playback resumes – otherwise playback fails.
Alternate Audio Rendition playlist	If content specified in the alternate track is also available in the main track (i.e. say the main track has audio but viewer wants to play alternate audio), the runtime simply ignores requests to change over to the alternate rendition. If the alternate rendition is the default audio track for the stream, then playback fails.

16. Handling Multiple Video Playback Controls

There may be cases where you may need to have more than one MediaElement (or <Video> tag) on a single application page, where each of them are playing HLS streams and are using this runtime to do so. In cases like that, one challenge that arises is to uniquely identify the ***IHLSController*** instance that is tied to the specific MediaElement(or <video> tag) instance.

As an example, if you have 3 MediaElement instances, each of which are pointing to HLS streams, you will get the ***ControllerReady*** event raised three times on the single ***ControllerFactory*** instance that you

had supplied while registering the byte stream handler. But there are no guarantees that the events are raised in any specific order, and hence you cannot use event order to tie the controller instances supplied back to the application code to the corresponding MediaElement instances. You may think that inspecting the playlist URL and using that to identify the right controller could be a way - until for some reason you have the same playlist playing back in multiple MediaElements, when that method does not work anymore.

The HLS runtime addresses this by allowing you to annotate the source playlist URL's with controller identifiers. You supply a controller identifier by appending a unique string to the end of the URL separated by the "##" separator. Note that there is no specific restriction on what that identifier string can be as long as it is unique among other such identifiers that you may be using on the same page for other MediaElements (or <video> tags). So if you had a source URL like below:

http://somedomain.com/media/someHLSStream.m3u8

and you would like to annotate it for a MediaElement named "meFirst", you could write code such as :

meFirst.Source = "http://somedomain.com/media/someHLSStream.m3u8##meFirst";

Note that we are using the MediaElement name – but to reiterate this could be any unique string that ties it to the control in question.

When the runtime finds an annotated URL like this, it strips the identifier back from the URL and supplies it back to your code through the ***IHLSController.ID*** property. You can easily inspect the ***ID*** property and use the value to tie a controller uniquely to the control that it is meant for.

17. Playlist Batching

Occasionally you might face a situation where a single piece of VOD content (single or multi bitrate) has been segmented into multiple smaller pieces of VOD clips. Usually this means that each such content part is represented with its own individual stand-alone playlist. However there may be a need to batch the constituent playlists such that to the viewer it seems that it is a single playlist playing with a single composite timeline – such that the viewer can seek back and forth across that timeline as well as bitrate switching works as it would if the source content was not segmented.

The HLS SDK provides an API to enable this feature. To use the feature you register the extension and set the source to the first of the multiple constituent playlists that you want to batch together. In the ***ControllerReady*** event handler, you call the ***Controller.BatchPlaylists()*** method and pass in an array of the playlist URL strings in playback order, starting from the second playlist URL (note that that you have already used the first URL to set the source property and hence should not include it in this array any more). Also note that if your content is multi bitrate, this feature will only work if each playlist is

also multi-bitrate and has the same number of bitrates across the range. That is all you will need to do and the HLS runtime internally merges the playlists and plays it back as a single continuous piece of content. Note that playlist batching is not available for live content.

17.1. Playlist batching and timestamps

Playlist batching works by modifying the actual timestamps associated with content samples in subsequent playlists beyond the first entry such that the playback engine thinks it is one continuous piece of content. If the content has "in-band" data other than audio and video that the HLS runtime recognizes and processes (like 608 closed captioning content or HLS timed metadata) it will manage the timestamps for those content types as well. However on occasions, you may have to deal with "out of band" content such as TTML closed captions etc. that the HLS runtime has not visibility into. If such content is also segmented following the same rules, then you may find that each segment may have timestamps that are relative to that specific segment and not to the overall timeline – and this will cause such content to fall out of sync with the playback as any timestamps in that content is not being actively modified by the HLS runtime. However in most cases it is fairly easy to do the modification yourself in player level code. Let's take a TTML markup segment as an example – if the TTML segment corresponds to the nth playlist segment in the batch that you are playing back, all you will need to do is to increment every timestamp (begin and end attributes in case of TTML) by the sum total duration of all playlists starting from 1 to (n-1). However to do this you will need to know the duration of each individual playlist in the batch. To facilitate this the HLS SDK exposes an API on the Playlist object called ***GetPlaylistBatchItemDurations()*** which returns an array of duration values (measured in ticks i.e. 100 nanosecond units) for all the individual playlists in the batch starting with and including the very first playlist.

18. Custom Downloaders

The HLS SDK allows you to inject custom developed downloader components into the pipeline. Custom downloader components can be used to enable many scenarios. An example could be using a custom downloader to store content on disk during playback and enable offline viewing later on. Another example could be enabling alternate mechanisms of key acquisition.

To implement a custom downloader you need to build a WinRT component that implements the Microsoft.HLSClient.IHLSContentDownloader interface. This interface is listed below.

```
public interface IHLSContentDownloader
{
  //used to indicate to a caller if an implementing type is currently performing a download
  bool IsBusy { get; }
  //used to indicate to a caller if an implementing type supports cancellation of a download
  bool SupportsCancellation { get; }
  //raised by an implementing type when a download completes successfully
  event TypedEventHandler<IHLSContentDownloader, IHLSContentDownloadCompletedArgs> Completed;
  //raised by an implementing type when a download fails
  event TypedEventHandler<IHLSContentDownloader, IHLSContentDownloadErrorArgs> Error;
```

```
  //when called the implementing type will cancel an ongoing download
  void Cancel();
  //when called an implementing type will start an asyncronous download operation
  IAsyncAction DownloadAsync();
  //called to supply an implementing type with the absolute URI of an asset to be downloaded
  void Initialize(string ContentUri);
}
```

The runtime passes in the URL of the asset it needs downloaded using the Initialize method. Following that it invokes the DownloadAsync() method where your implementation should perform the actual task acquiring the asset content. Once you successfully acquire the asset, you should raise the Completed event. If there is a failure, you should raise the Error event. While you are inside the DownloadAsync() method you should return true for the IsBusy property. If you support a cancellable download, you should return true for the SupportsCancellation property and be prepared to handle a cancellation when the runtime calls the Cancel() method. Supporting cancellation is not a strict requirement.

To support the Completed and Error events, you will also need to author two additional WinRT types that implement IHLSContentDownloadCompletedArgs and IHLSContentDownloadErrorArgs interfaces respectively. The former is used to pass on the actual downloaded content vi the Content property, as well the URL of the content, the HttpStatusCode (if your acquisition does not involve going over Http – set this to HttpStatusCode.Ok), and any Http response headers if applicable. In case of an error, you simply set the error statuc code.

```
public interface IHLSContentDownloadCompletedArgs
{
  //the downloaded content
  IBuffer Content { get; }
  //the absolute URI of the asset
  Uri ContentUri { get; }
  //Does the status code indicate success ?
  bool IsSuccessStatusCode { get; }
  //Any response headers doenloaded along with the content
  IReadOnlyDictionary<string, string> ResponseHeaders { get; }
  //The download status code
  HttpStatusCode StatusCode { get; }
}



public interface IHLSContentDownloadErrorArgs
{
  //The actual error code
  HttpStatusCode StatusCode { get; }
}
```

To supply a custom downloader to the runtime, you use the resource request interception technique discussed in section 13.4. The IHLSResourceRequestEventArgs.SetDownloader() method can be called in your PrepareResourceRequest event handler to pass in an instance of your custom downloader type to the runtime.

Note that since the HLS runtime can conduct multiple downloads concurrently, it is a good practice to not share a single instance of a custom downloader type across multiple download requests. To the extent possible you should make your downloader types stateless, and any shared state that needs to be maintained should be maintained outside the downloader in app level code. Writing blocking code in the DownloadAsync() method in an attempt to serialize calls to a shared downloader instance can severely degrade performance.

19. Using the Microsoft Media Platform Player Framework (MMPPF) Plugins for HLS

19.1. One-Time Setup

- Download and install the following:
  - *Microsoft Windows and Windows Phone 8.1 Adaptive Streaming SDK*
  - *Microsoft Player Framework*
  - *Microsoft HLS PF Plugins (installed from the MSHLS.PF.vsix file.)*

19.2. Using the HLS Plugin

19.2.1. Quick Start for JavaScript

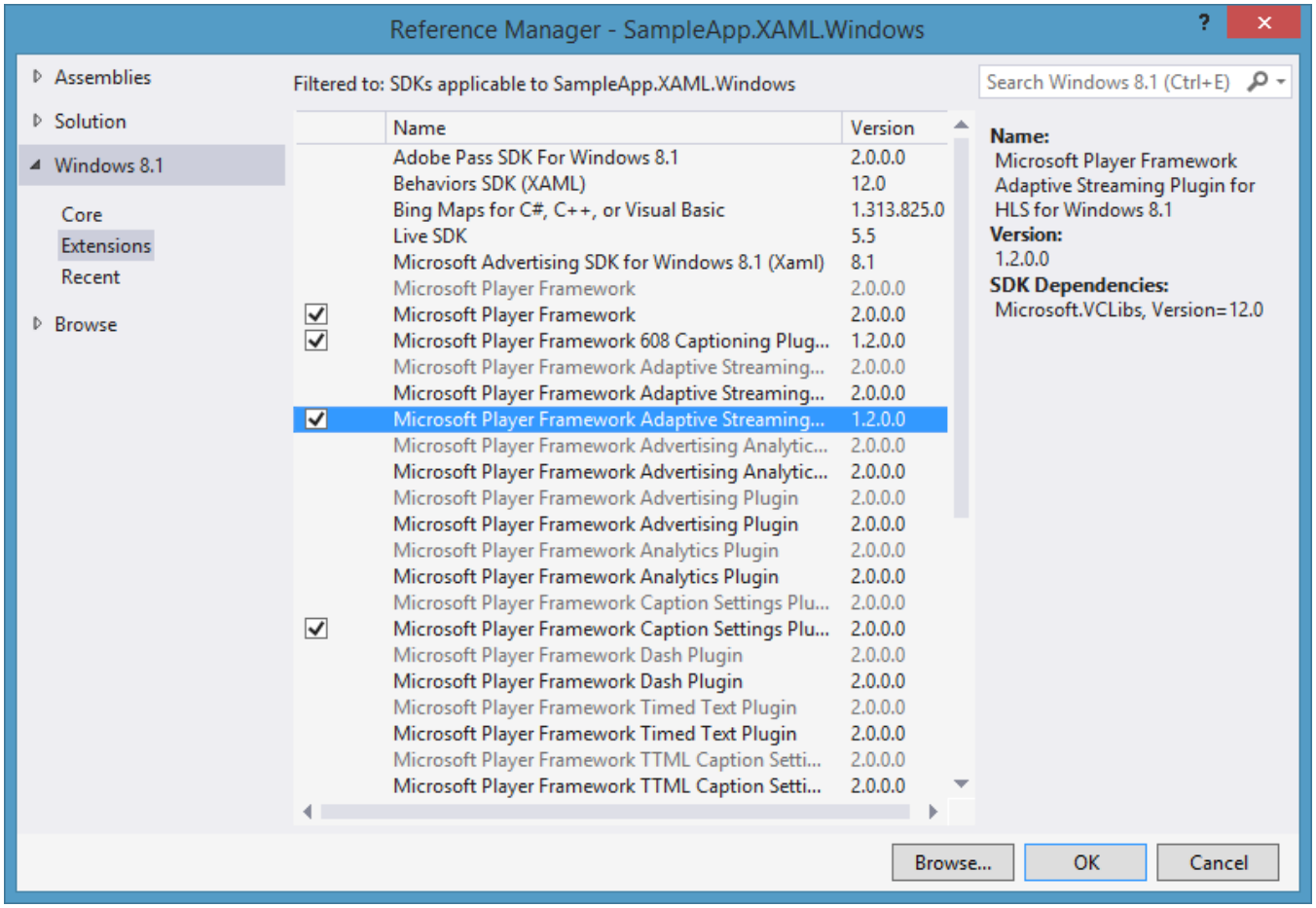
This plugin adds additional Player Framework and infrastructure support for the Microsoft Windows HLS SDK. Specifically, this plugin:

- Registers the HLS SDK into the Media Foundation pipeline, so that setting the source of the media player to an HLS playlist file URL will start video playback.
- Directly supports additional registrations for non-standard playlist file extensions or MIME types.
- Wires up the audio renditions available in the presentation into the Player Framework's audio track control.
- Updates the signal strength and media quality displays of the Player Framework's player based on data from the SDK.
- Exposes some important events and properties from the SDK directly to JavaScript.
- Supports extraction of 608 captioning data from the SDK and ferries the raw caption data to the Player Framework, so that the 608 captioning component can process it. The plugin's closedCaptionType property must be set to ClosedCaptionType.CC608Instream (1) for 608 processing to occur.

- Exposes an instream608captionsavailable event that is raised when 608 data is received (and the plugin's closedCaptionType property has been set to 1). This can be used by the app to make the caption track controls visible for a stream where the app is uncertain if it has 608 data or not.
- Exposes the controller and the controller factory from the SDK (via the hlscontrollerready event and the controllerFactory property) to JavaScript code for more advanced scenarios

19.2.1.1. Initial App Creation

- Create a new JS Windows Store app project in Visual Studio
- Select a specific project architecture (x64, x86, ARM) for the project using the Configuration Manager:
    - In Visual Studio, click **BUILD**, then **Configuration Manager**
    - Select the desired **Active Solution Platform** in the upper right corner
- Right-click the References folder in the Visual Studio project, click **Add Reference…**, and add the following references from the Windows > Extensions pane:
    - Microsoft Player Framework
    - For Windows apps: Microsoft Windows 8.1 Adaptive Streaming SDK
    - For Windows apps: Microsoft Player Framework Adaptive Streaming Plugin for HLS for Windows 8.1
    - For Windows Phone apps: Microsoft Windows Phone 8.1 Adaptive Streaming SDK
    - For Windows Phone apps: Microsoft Player Framework Adaptive Streaming Plugin for HLS for Windows Phone 8.1



- Add references to the markup:
    - Open the outer HTML page for the app
    - Expand the Microsoft Player Framework reference and drag the js file and the desired CSS file to the markup between the WinJS references and the app-level references in the outer HTML page
    - Add the Player Framework and HLS Plugin references like so:

```
<!-- Player Framework References -->
<link href="/Microsoft.PlayerFramework.Js/css/PlayerFramework-dark.css"/>
<script src="/Microsoft.PlayerFramework.Js/js/PlayerFramework.js"></script>

<!-- HLS Plugin References -->
<script src="/Microsoft.PlayerFramework.Js.HLS/js/HLSPlugin.js"></script>
```

- Add the following to the package.appmanifest file for the application near the bottom. (This registers the necessary WinRT component from the HLS SDK.)

```
  <Extensions>
    <Extension Category="windows.activatableClass.inProcessServer">
      <InProcessServer>
        <Path>Microsoft.HLSClient.dll</Path>
        <ActivatableClass ActivatableClassId="Microsoft.HLSClient.HLSPlaylistHandler" Th
readingModel="both" />
      </InProcessServer>
    </Extension>

  </Extensions>
```

- Add the following markup for the Player Framework Player to the media page in the app: (it includes a link to Apple's Advanced HLS Test Stream)

```
<div data-win-control="PlayerFramework.MediaPlayer" data-win-options="{
                width: 640,
                height: 360,
                autoplay: true,
                src:
'https://devimages.apple.com.edgekey.net/streaming/examples/bipbop_16x9/bipbop_16x9_variant.m
3u8'
            }"></div>
```

- At this point, pressing F5 in Visual Studio should build and launch the app, and the video should play. The Apple Advanced HLS Test Stream should begin to play automatically.

19.2.1.2. Plugin API

The plugin automatically handles default registration of the HLS components into the Media Foundation pipeline, and provides default integration with the Player Framework. Additional functionality is exposed via the JavaScript API.

Properties

Name	Type	Purpose
startupBitrate	Integer (bits per sec)	When set before the video starts, this assists the system to determine which bitrate it should begin playback with. After playback

		starts, prevailing network conditions determine which bitrates to use.
maxBitrate	Integer (bits per sec)	Sets a maximum bitrate—the system will not attempt to shift to streams that have a higher bandwidth than maxBitrate.
minBitrate	Integer (bits per sec)	Sets a minimum bitrate—the system will not attempt to shift to streams that have a lower bandwidth than minBitrate.
closedCaptionType	ClosedCaptionType enum: 0 (None) 1 (CC608Instream)	Configures the plugin to process 608 in-stream data or not.
controllerFactory	IHLSControllerFactory	Exposes the HLS SDK's controller factory instance. This can be used for interception of the initial playlist HTTP request.

Warning: Setting any of the bitrate properties constrains the normal functionality which allows the system to pick the stream best suited to current network conditions. Therefore, setting too low a maxBitrate may result in low quality playback, while setting too high a minBitrate may mean that playback becomes dominated by buffering in situations with relatively low network bandwidth.

Methods

The API offers only a single method. The method is used for registering non-standard MIME types and file extensions for the playlist file. (For example, some server-side products may serve the playlist with a non-standard MIME type that cannot be easily changed.)

In order to be effective, this method must be called before the video starts.

Note that the standard file extension (".m3u8") and MIME types ("application/vnd.apple.mpegurl" and "application/x-mpegurl") are automatically registered by the plugin—the method below need only be used if these standard values will not be used.

Name	Purpose	Parameters
addNonStandardRegistration	If the MIME type or file extension of the playlist files are non-standard, this method (called before the video is started) registers the non-standard types so they are efficiently processed by the system. (See below)	• **fileExtension**: string parameter with the file extension of the playlist file. It should include the period, for example: ".txt" • **mimeType**: string parameter with the

		MIME type for the playlist file

Note that due to the way the Media Foundation pipeline works, video will play regardless of whether the correct registration is used or not. If Media Foundation does not find any components that support the actual file extension and MIME type, it iterates through all possible media playing components, trying each one and seeing if it is able to play video or not. Therefore, eventually the pipeline will find the proper component if it has any registrations. The purpose of this method is to make the process much quicker and more efficient by empowering the pipeline to choose the correct component first.

In short, if video plays but it takes a long time than usual to startup, it's likely that the pipeline registration may not be correct.

Events

Name	When It Fires	Event Arguments
hlscontrollerready	Fires each time video playback begins, and allows listeners to get a reference to the controller for advanced scenarios. (See below.)	• **detail** contains a reference to the controller instance.

bitrateswitchsuggestedcannotcancel	Fires when the system decides it should switch to a higher or lower bandwidth stream. This event cannot be cancelled from JavaScript—if this is desired, you will need to write a custom WinRT component or use a different language.	On the detail property:
		• **fromBitrate** provides the bitrate to switch from (in bits per second)
		• **toBitrate** provides the bitrate to switch to (in bits per second)
		• **forTrackType** indicates which track type is switching:
		  ○ VIDEO: 0
		  ○ AUDIO: 1
		  ○ BOTH: 2
		• **cancel** is not usable from JavaScript

bitrateswitchcompleted	Fires when the switch to a different bandwidth stream has completed	Same as above—provides information about the switch that completed
bitrateswitchcancelled	Fires when a pending bitrate switch is cancelled (either due to a cancel request from a	Same as above—provides information about the switch which was cancelled

	listener, or due to changing network conditions)	

streamselectionchanged	Fires when the stream selection changes. For example, if the system switches from a stream with both audio and video to a lower-bandwidth stream with only audio, this event will fire to inform the app that video will no longer be available. Similarly, when switching from audio-only to a stream with both audio and video, this event fires to let the app know that video is now available.
	
	A common usage of this event would be to hide the video when in audio-only mode and perhaps alert the user that, due to network bandwidth limitations, they are currently getting audio-only.
		On the detail property:
		• **from** property provides which track type it was playing previously:
		  ○ VIDEO: 0
		  ○ AUDIO: 1
		  ○ BOTH: 2
		• **to** property provides which track type is now starting (same values as above). For example, if the system switches from both (audio and video) to audio-only, then **from** = 2 and **to** = 1.

instream608captionsavailable	Fires when 608 data is extracted from the stream, and the closedCaptionType is set to CC608Instream (1). A common usage of this event would be to show the captioning controls for a stream where it was unclear if there would be 608 data available or not. Note that this event will fire each time 608 data is extracted—this will generally happen once per segment played.	Empty object.

availablecaptionspopulated	Fires when the plugin has populated the caption tracks with "CC1", "CC2", "CC3", and "CC4". A common usage of this event would be to customize the names of the tracks to something more meaningful (for example "English" or "Spanish"), and to delete caption track objects that have no associated caption data. Another possible usage of this event would be to delete the supplied caption objects and inject your own custom objects. Note that the objects and their id values are used internally by the captioning framework—therefore, if you add any new objects (as opposed to renaming the existing objects), you will be responsible for making them function correctly.	Empty object

19.2.1.3. Advanced Scenarios

The JavaScript Player Framework plugin allows access to the HLS Controller class via the hlscontrollerready event. Using this event, you can directly access the HLS Controller class when it is created, and then use the rich API directly for advanced scenarios. In addition, the controller factory is also exposed via the controllerFactory property of the plugin.

19.2.2. Quick Start for XAML/C#

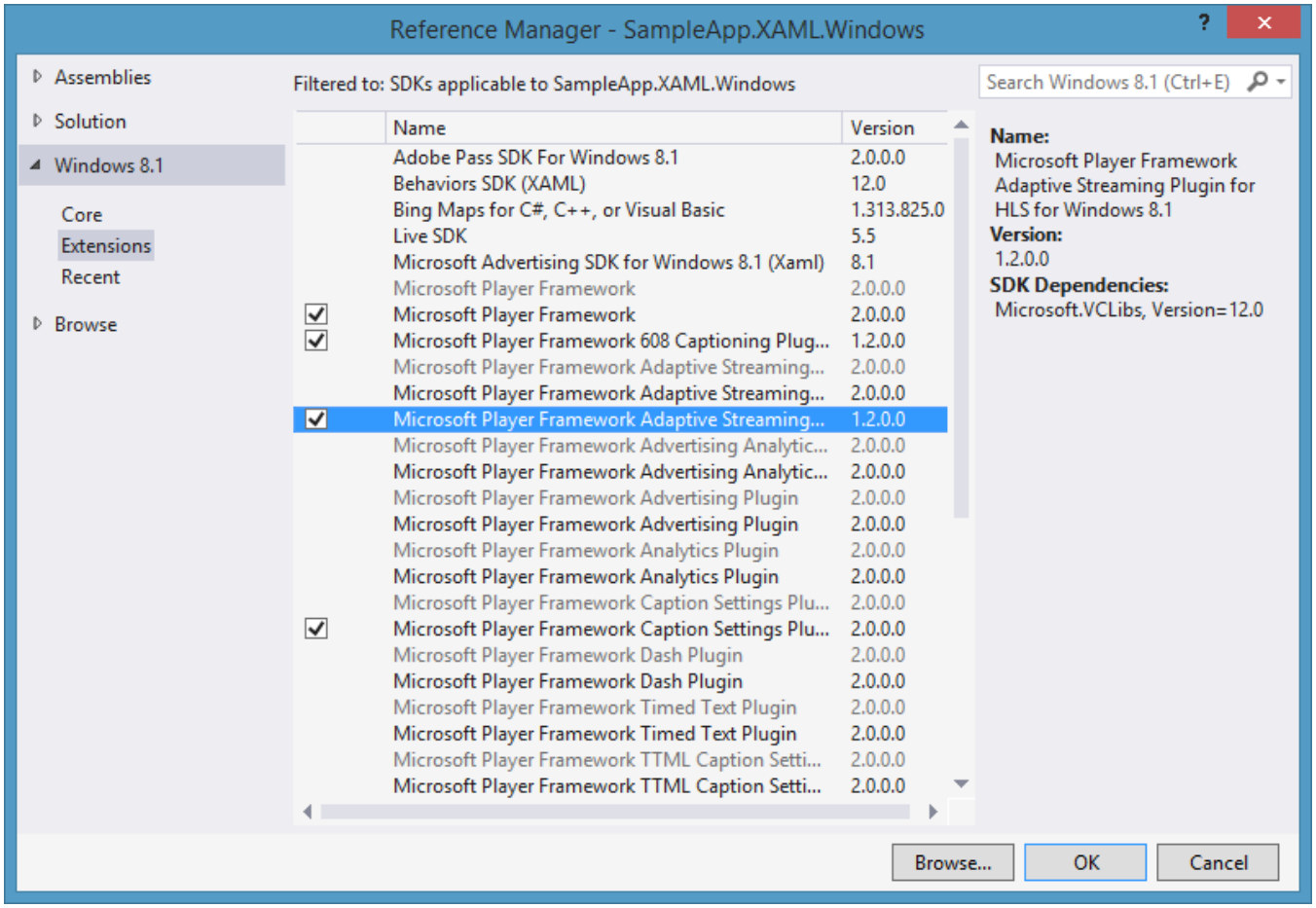
This plugin adds additional Player Framework and infrastructure support for the Microsoft Windows HLS SDK. Specifically, this plugin:

- Registers the HLS SDK into the Media Foundation pipeline, so that setting the source of the media player to an HLS playlist file URL will start video playback.

- Directly supports additional registrations for non-standard playlist file extensions or MIME types.
- Wires up the audio renditions available in the presentation into the Player Framework's audio track control. (When the audio renditions are supported by the SDK.)
- Updates the signal strength and media quality displays of the Player Framework based on data from the SDK.
- Exposes some important events and properties from the SDK directly to C# and XAML.
- Supports extraction of in-stream 608 captioning data from the SDK and ferries the raw caption data to the Player Framework, so that the 608 captioning component can process it. (To do this, you need to set the plugin's ClosedCaptionType property to ClosedCaptionType.CC608Instream.)
- Exposes the Instream608CaptionsAvailable event, which is raised when 608 data is extracted (and the ClosedCaptionType property on the plugin is set to CC608Instream). This could be useful for knowing when to show the caption track controls for a stream where the app doesn't know if it will contain 608 captions or not.
- Supports downloading of WebVTT captioning data and ferries the raw caption data to the Player Framework, so that the WebVTT captioning component can process it. (To do this, you need to set the plugin's ClosedCaptionType property to ClosedCaptionType.WebVTTSidecar.)
- Exposes the controller and controller factory objects from the SDK (via the HLSControllerReady event and the ControllerFactory property) to C# code for advanced scenarios

19.2.2.1. Initial App Creation

- Create a new Visual C# Windows Store app project in Visual Studio
- Select a specific project architecture (x64, x86, ARM) for the project using the Configuration Manager:
    - In Visual Studio, click **BUILD**, then **Configuration Manager**
    - Select the desired **Active Solution Platform** in the upper right corner
- Right-click the References folder in the Visual Studio project, click **Add Reference…**, and add the following references from the Windows > Extensions pane:
    - Microsoft Player Framework
    - For Windows apps: Microsoft Windows 8.1 Adaptive Streaming SDK
    - For Windows apps: Microsoft Player Framework Adaptive Streaming Plugin for HLS for Windows 8.1
    - For Windows Phone apps: Microsoft Windows Phone 8.1 Adaptive Streaming SDK
    - For Windows Phone apps: Microsoft Player Framework Adaptive Streaming Plugin for HLS for Windows Phone 8.1



- Add references to the markup:
- In the XAML view with the media player, add the following namespaces:

```
xmlns:hls="using:Microsoft.PlayerFramework.Adaptive.HLS"
xmlns:mmppf="using:Microsoft.PlayerFramework"
```

- Add a simple MediaPlayer control with the HLS Plugin to the view:

```
<mmppf:MediaPlayer
    x:Name="player"
    IsSkipPreviousVisible="False"
    IsSkipNextVisible="False"
    AutoPlay="True"
    Width="1068"
    Height="600"
    HorizontalAlignment="Left"
    VerticalAlignment="Top">
    <mmppf:MediaPlayer.Plugins>
        <hls:HLSPlugin />
    </mmppf:MediaPlayer.Plugins>
</mmppf:MediaPlayer>
```

When finished, it should look something like this:

```
<Page
    x:Class="TestXamlVsix.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:TestXamlVsix"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:hls="using:Microsoft.PlayerFramework.Adaptive.HLS"
    xmlns:mmppf="using:Microsoft.PlayerFramework"

    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <mmppf:MediaPlayer
            x:Name="player"
            IsSkipPreviousVisible="False"
            IsSkipNextVisible="False"
            AutoPlay="True"
            Width="1068"
            Height="600"
            HorizontalAlignment="Left"
            VerticalAlignment="Top">
            <mmppf:MediaPlayer.Plugins>
                <hls:HLSPlugin />
            </mmppf:MediaPlayer.Plugins>
        </mmppf:MediaPlayer>

    </Grid>
</Page>
```

- Add the following to the Package.appmanifest file for the application near the bottom. (This registers the necessary component from the HLS SDK.)

```
  <Extensions>
    <Extension Category="windows.activatableClass.inProcessServer">
      <InProcessServer>
        <Path>Microsoft.HLSClient.dll</Path>
        <ActivatableClass ActivatableClassId="Microsoft.HLSClient.HLSPlaylistHandler"
ThreadingModel="both" />
      </InProcessServer>
    </Extension>
  </Extensions>
```

- Add the following line of code to the code-behind file of the view, in the OnNavigatedTo method:

```
player.Source = new
Uri("https://devimages.apple.com.edgekey.net/streaming/examples/bipbop_4x3/bipbop_4x3_variant
.m3u8");
```

- At this point, pressing F5 in Visual Studio should build and launch the app, and the video should play. The Apple Simple HLS Test Stream should begin to play automatically.

19.2.2.2. Plugin API

The plugin automatically handles default registration of the HLS components into the Media Foundation pipeline, and provides default integration with the Player Framework. Additional functionality is exposed via the C# API.

Properties

Name	Type	Purpose
StartupBitrate	Nullable unsigned int (bits per sec)	When set before the video starts, this assists the system to determine which bitrate it should begin playback with. After playback starts, prevailing network conditions determine which bitrates to use.
MaxBitrate	Nullable unsigned int (bits per sec)	Sets a maximum bitrate—the system will not attempt to shift to streams that have a higher bandwidth than MaxBitrate.
MinBitrate	Nullable unsigned int (bits per sec)	Sets a minimum bitrate—the system will not attempt to shift to streams that have a lower bandwidth than MinBitrate.
ClosedCaptionType	ClosedCaptionType enum	Configures the plugin to process data for the specified closed caption type. Default is ClosedCaptionType.None. Current options include ClosedCaptionType.WebVTTSidecar, and ClosedCaptionType.CC608Instream.
ControllerFactory	IHLSControllerFactory	Provides access to the HLS SDK's instance of the ControllerFactory. This can be used in advanced scenarios to intercept the initial playlist HTTP request for reasons such as adding security tokens.

Warning: Setting any of the bitrate properties constrains the normal functionality which allows the system to pick the stream best suited to current network conditions. Therefore, setting too low a maxBitrate may result in low quality playback, while setting too high a minBitrate may mean that playback becomes dominated by buffering in situations with relatively low network bandwidth.

Methods

The API offers only a single method. This method is used for registering non-standard MIME types and file extensions for the playlist file. (For example, some server-side products may serve the playlist with a non-standard MIME type that cannot be easily changed.)

In order to be effective, this method must be called before the video starts.

Note that the standard file extension (".m3u8") and MIME types ("application/vnd.apple.mpegurl" and "application/x-mpegurl") are automatically registered by the plugin—the method below need only be used if these standard values will not be used.

Name	Purpose	Parameters
AddNonStandardRegistration	If the MIME type or file extension of the playlist files are non-standard, this method (called before the video is started) registers the non-standard types so they are efficiently processed by the system. (See below)	• **fileExtension**: string parameter with the file extension of the playlist file. It should include the period, for example: ".txt" • **mimeType**: string parameter with the MIME type for the playlist file

Note that due to the way the Media Foundation pipeline works, video will play regardless of whether the correct registration is used or not. If Media Foundation does not find any components that support the actual file extension and MIME type, it iterates through all possible media playing components, trying each one and seeing if it is able to play video or not. Therefore, eventually the pipeline will find the proper component if it has any registrations. The purpose of this method is to make the process much quicker and more efficient by empowering the pipeline to choose the correct component first.

In short, if video plays but it takes a long time than usual to startup, it's likely that the pipeline registration may not be correct.

Events

Name	When It Fires	Event Arguments
HLSControllerReady	Fires each time video playback begins, and allows listeners to get a reference to the controller for advanced scenarios. (See below.)	The event args parameter is a reference to the controller instance.

BitrateSwitchSuggested	Fires when the system decides it should switch to a higher or lower bandwidth stream. Can cancel the bitrate from switching by setting args.Cancel to true.	On the event args parameter:
		• **FromBitrate:** provides the bitrate to switch from (in bits per second)
		• **ToBitrate:** provides the bitrate to switch to (in bits per second)
		• **ForTrackType:** indicates which track type is switching:
		  ○ VIDEO
		  ○ AUDIO
		  ○ BOTH
		• **Cancel:** if set to true in the body of the event handler synchronously (not dispatched to the UI thread), then the bitrate switch event will be cancelled.

BitrateSwitchCompleted	Fires when the switch to a different bandwidth stream has completed.	Same as above—provides information about the switch that completed.
BitrateSwitchCancelled	Fires when a pending bitrate switch is cancelled (either due to a cancel request from a listener, or due to changing network conditions).	Same as above—provides information about the switch which was cancelled.

StreamSelectionChanged	Fires when the stream selection changes. For example, if the system switches from a stream with both audio and video to a lower-bandwidth stream with only audio, this event will fire to inform the app that video will no longer be available. Similarly, when switching from audio-only to a stream with both audio and video, this event fires to let the app know that video is now available.	On the event args parameter:
		• **From**: indicates which track type was playing previously:
		  ○ VIDEO
		  ○ AUDIO
		  ○ BOTH
		• **To**: indicates which track type is starting now (same values as above). For example, if the system switches from both (audio and video) to audio-only, then From = 2 and To = 1.

	A common usage of this event would be to hide the video when in audio-only mode and perhaps alert the user that, due to network bandwidth limitations, they are currently getting audio-only.	
AvailableCaptionsPopulated	Fires after the plugin populates either WebVTT or 608 caption tracks. A common usage of this event would be to change the default 608 caption track names ("CC1", "CC2", "CC3", and "CC4") to more meaningful names for the app, and hide any caption tracks that are known to not contain data.	EventArgs.Empty
Instream608CaptionsAvailable	Fires when 608 data has been extracted from the stream, and the ClosedCaptionType property is set to ClosedCaptionType. CC608Instream	EventArgs.Empty

19.2.2.3. Advanced Scenarios

The Xaml Player Framework plugin allows access to the HLS Controller class via the HLSControllerReady event. Using this event, you can directly access the HLS Controller class when it is created, and then use the rich API available through the controller reference directly for advanced scenarios.

19.3. Using the 608 Closed Captioning Plugin

19.3.1. Quick Start for JavaScript

This plugin adds the ability to display in-stream 608 captions for supported video types for Microsoft Windows Store apps. (Currently, only HLS video is supported.) Specifically:

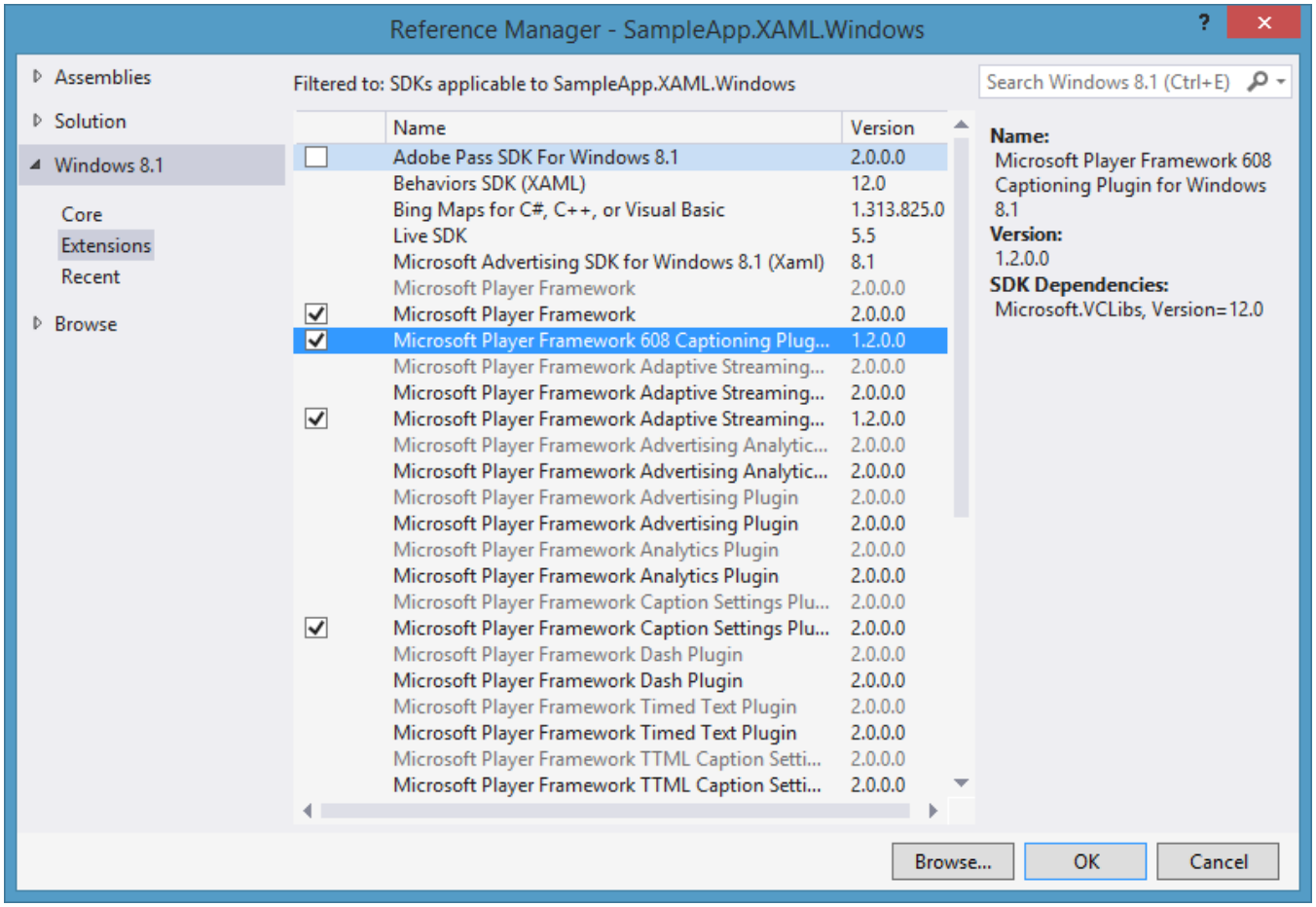
- It interacts with the *HLS SDK* to support in-stream 608 captioning.
- It exposes a *currentTrack* property to programmatically select which 608 caption track is displayed.

- It exposes *show* and *hide* methods (to show and hide the captions).

19.3.1.1. Initial App Creation

- Follow the steps for initial app creation from the MMPPF HLS plugin section.
- Right-click the **References** folder in the Visual Studio project, click **Add Reference…**, and add the following reference from the Windows > Extensions pane:
  - Microsoft Player Framework 608 Captioning Plugin for Windows 8.1

It should look something like this when completed:



- Add additional references to the markup:
  - Open the outer HTML page for the app
  - Add a reference to the CC608Plugin.js and CC608Plugin.css like so:

```
<!-- HLS Plugin Reference -->
<script src="/Microsoft.PlayerFramework.Js.HLS/js/HLSPlugin.js"></script>
<link href="/Microsoft.PlayerFramework.Js.CC608/css/CC608Plugin.css" rel="stylesheet">
<script src="/Microsoft.PlayerFramework.Js.CC608/js/CC608Plugin.js"></script>
```

- Replace the existing video tag with the following markup in the media page in the app. Note that you MUST set the closedCaptionType to 1 for the HLS plugin—this activates the 608 caption logic!

```
<div class="player" id="player" data-win-control="PlayerFramework.MediaPlayer" data-win-
options="{
    autoplay: true,
    isCaptionsVisible: true,
    width: 1024,
    height: 768,
    src:
'http://devimages.apple.com.edgekey.net/streaming/examples/bipbop_16x9/bipbop_16x9_variant.m3u8'
,
    hLSPlugin: { closedCaptionType: 1 }
}">
</div>
```

- At this point, pressing F5 in Visual Studio should build and launch the app, and the video should play. The Apple Advanced HLS Test Stream should begin to play automatically. You can then select the CC1 caption track, and the captions will begin to display around 11 seconds into the stream.

19.3.1.2. Plugin API

The plugin automatically handles setup of captioning, and by default the current caption track is wired up to the caption track selection control.

If you need to set the track programmatically, the API contains a property for the current track. The object passed to the currentTrack property must be:

- In the mediaPlayer.captionTracks collection
- An instance of PlayerFramework.DynamicTextTrack
- Have a _stream.id value of 0, 1, 2, 3, or 4

Properties

Name	Type	Purpose
currentTrack	PlayerFramework. DynamicTextTrack	Gets or sets the caption track to be displayed, based on the _stream.id value: • 0: No captions • 1: CC1 • 2: CC2 • 3: CC3 • 4: CC4

Methods

Name	Purpose
show()	Shows the captions. This is the default mode.

hide()	Hides the captions.

Events

Name	When it fires	Event Arguments
currentcaptiontrackchange	Fires when the caption track changes.	None

To customize the captions display, please see the documentation for the MMPPF HLS plugin earlier—there is an event called availablecaptionspopulated, and the application can use this event to modify the existing caption objects. For example, "CC1" could be changed to "English", "CC2" could be changed to "Spanish", and CC3 and CC4 could be removed because they do not contain caption data.

19.3.2. Quick Start for XAML/C#

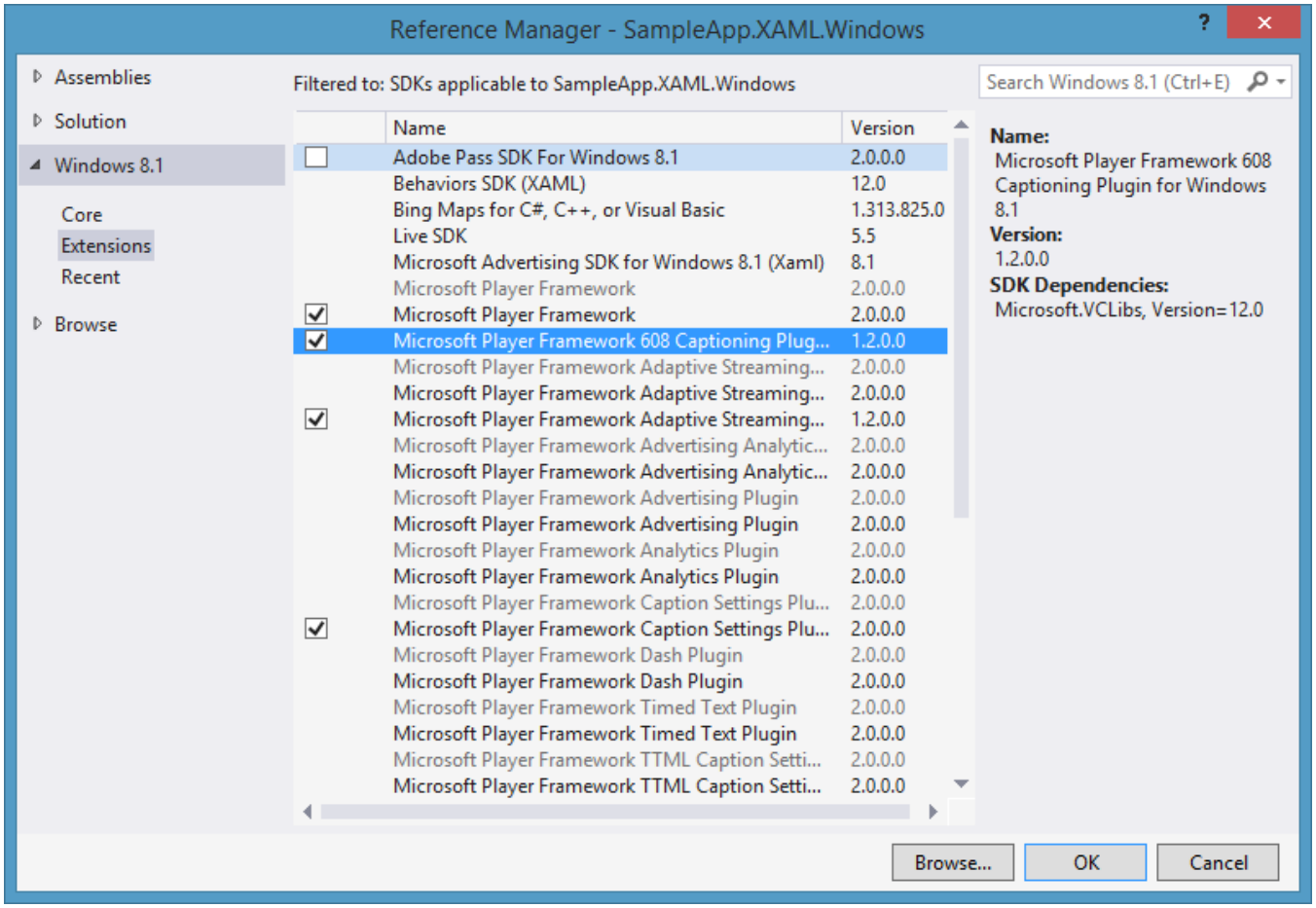
This plugin adds the ability to display in-stream 608 captions for supported video types for Microsoft Windows store apps. (Currently, only HLS video is supported.) Specifically:

- It interacts with the *HLS SDK* to support in-stream 608 captioning.
- It exposes a *CurrentTrack* property to programmatically select which 608 caption track is displayed.

19.3.2.1. Initial App Creation

- Follow the initial app creation steps from the MMPPF HLS plugin section
- Right-click the **References** folder in the Visual Studio project, click **Add Reference…**, and add the following references from the Windows > Extensions pane:
    - Microsoft Player Framework 608 Captioning Plugin for Windows 8.1

  It should look something like this when completed:



- In the XAML view with the media player, add the following namespace:

```
xmlns:cc608="using:Microsoft.PlayerFramework.CC608"
```

- Replace the existing MediaPlayer markup in the Xaml file with the following:

```
<mmppf:MediaPlayer
    x:Name="player"
    IsSkipPreviousVisible="False"
    IsSkipNextVisible="False"
    IsCaptionSelectionVisible="True"
    AutoPlay="True"
    Width="1068"
    Height="600"
    HorizontalAlignment="Left"
    VerticalAlignment="Top">
    <mmppf:MediaPlayer.Plugins>
        <hls:HLSPlugin ClosedCaptionType="CC608Instream" />
        <cc608:CC608Plugin />
    </mmppf:MediaPlayer.Plugins>
</mmppf:MediaPlayer>
```

This includes the CC608 plugin, configures the HLS plugin for 608 captions, and sets the IsCaptionSelectionVisible to true to allow the user to select a caption track.

When finished, it should look something like this:

```
<Page
    x:Class="TestXamlApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:TestXamlApp"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:hls="using:Microsoft.PlayerFramework.Adaptive.HLS"
    xmlns:mmppf="using:Microsoft.PlayerFramework"
    xmlns:cc608="using:Microsoft.PlayerFramework.CC608"
    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <mmppf:MediaPlayer
            x:Name="player"
            IsSkipPreviousVisible="False"
            IsSkipNextVisible="False"
            IsCaptionSelectionVisible="True"
            AutoPlay="True"
            Width="1068"
            Height="600"
            HorizontalAlignment="Left"
            VerticalAlignment="Top">
            <mmppf:MediaPlayer.Plugins>
                <hls:HLSPlugin ClosedCaptionType="CC608Instream" />
                <cc608:CC608Plugin />
            </mmppf:MediaPlayer.Plugins>
        </mmppf:MediaPlayer>

    </Grid>
</Page>
```

- At this point, pressing F5 in Visual Studio should build and launch the app, and the video should play. The Apple Simple HLS Test Stream should begin to play automatically. Selecting CC1 in the captions selection control will display the 608 captions, which start around 11 seconds into the stream.

19.3.2.2. Plugin API

The plugin automatically handles setup of captioning, and by default the current caption track is wired up to the caption track selection control.

If you need to set the track programmatically, the API contains a property for the current track. The object passed to the *CurrentTrack* property must be:

- In the MediaPlayer.AvailableCaptions collection
- An instance of Microsoft.PlayerFramework.Caption

- Have an Id value of "1" for CC1, "2" for CC2, etc

Properties

Name	Type	Purpose
CurrentTrack	Microsoft. PlayerFramework. Caption	Gets or sets the caption track to be displayed, based on the Id value: • "1": CC1 • "2": CC2 • "3": CC3 • "4": CC4

To customize the captions display, please see the documentation for the MMPPF HLS plugin —there is an event called AvailableCaptionsPopulated, and the application can use this event to modify the existing caption objects. For example, "CC1" could be changed to "English", "CC2" could be changed to "Spanish", and CC3 and CC4 could be removed because they do not contain caption data.