



Adobe HTTP Dynamic Streaming SDK Broadcast Sample Applications

3.5

Contents

Overview	5
OfflinePackager	5
Input and output file formats	5
Segments and fragments	6
Packaging content without encryption	6
On Microsoft® Windows®	6
On Linux	6
Packaging and encrypting content	7
Command-line options for fragmenting files	8
Configure Protected HTTP Dynamic Streaming (PHDS)	9
Requesting PHDS certificates	10
The OfflinePackager configuration file	10
Command-line options for encrypting files	11
Command-line options for license rotation	14
Command-line options for blackout	14
The OfflinePackager design	15
Packager design	16
License Rotation	17
Changes in HDS manifest for license rotation (New in 3.2)	19
New element <drmAdditionalHeaderSet>	19
Modification in <drmAdditionalHeader> element	20
Modification in <media> element	20
Regional blackout	21
Trick Play support in HDKB	22
External CEK support in HDKB (New in 3.5)	24
EncryptorApp	24

EncryptorApp design.....	25
MPEGParser	26
IMpegParser	26
IMpegParserDelegate	27
Considerations for using MpegParser.....	29
MPEGParser - Support for Closed Captioning (New in 3.0).....	30
The onCaptionInfo message.....	30
Generating onCaptionInfo messages.....	30
API changes	31
MPEGParser - Support for Ad signalling (New in 3.0).....	32
API changes	32
M3U8Generator	34
M3U8Generator - Support for HLS Ad signalling (New in 3.0)	36
M3U8Generator changes for HLS License Rotation support (New in 3.2)	38
M3U8Generator usage workflow	40
Step 1: Pre-encrypting content.....	40
Step 2: Adding Adobe Access Metadata	40
Sample code.....	43

Overview

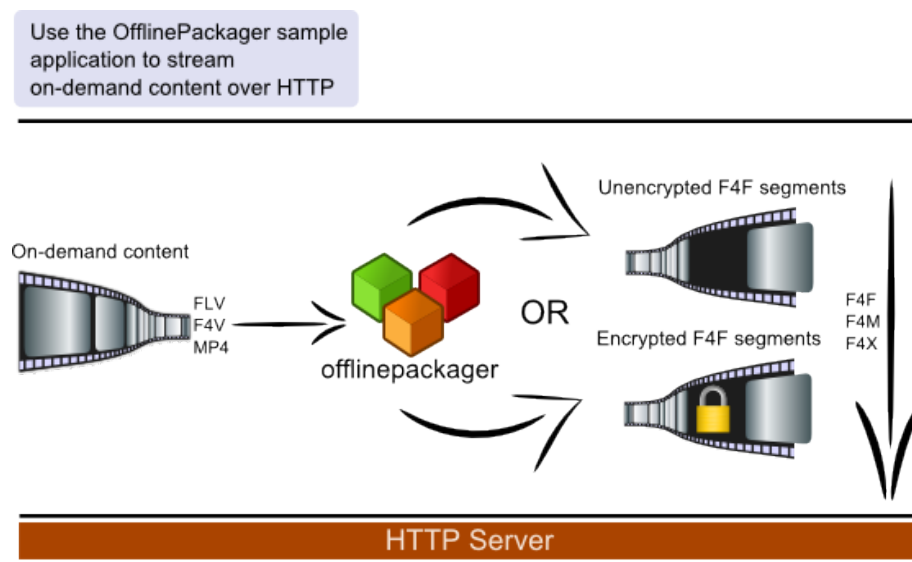
This document describes the following Adobe HTTP Dynamic Streaming SDK Broadcast(HDKB) sample applications:

- [OfflinePackager](#) (Enhanced with Trick Play support in HDKB 3.0)
- [EncryptorApp](#)
- [MPEGParser](#) (Enhanced with Ad Signalling and Closed Captioning support in HDKB 3.0)
- [M3U8Generator](#) (Enhanced with Ad Signalling support in HDKB 3.0)

OfflinePackager

The OfflinePackager is a command line tool that packages on-demand media files into fragmented video content that can be streamed over HTTP. The OfflinePackager can also encrypt files for protection with Adobe Access.

Figure 1: OfflinePackager



Input and output file formats

To stream on-demand media over HTTP, process the media file with the OfflinePackager. The OfflinePackager parses the content and packages it into F4F segments and fragments. You can also choose to have the OfflinePackager encrypt the content for use with Adobe Access.

OfflinePackager supports fragmentation of source files that have any of the following formats:

- FLV
- F4V
- MP4
- MPEG - 2 TS
- Adobe Access encrypted F4V/MP4

The OfflinePackager generates the following files:

File type	Description
.f4f	A stream segment. The tool generates one or more F4F files as output. Each file includes a segment of the source file. A segment contains one or more fragments of content. A player can use a URL to address each fragment.
.f4x	Index file. Contains the location of specific fragments within a stream.
.f4m	Flash Media Manifest file. Includes information about codec, resolution, and the availability of multi-bitrate files.

Segments and fragments

An f4f file contains a segment of a media file. Each segment can contain multiple fragments. When packaging a media file, use the `--segment-duration` and `--fragment-duration` options to set the length of each segment and fragment, in seconds. A fragment is appropriately compact to be sent over the Internet. However, it is sufficiently long to prevent excessive calls to the HTTP Server.

Packaging content without encryption

To package a file, pass the name of the input file to the OfflinePackager.

Specify the filename in one of the following ways:

- Enter the `--input-file` option at the command line.
- Create a configuration file that specifies the input file and enter the `--conf-file` option at the command line.

On Microsoft® Windows®

1. Open a Command window.
2. Change directories to the directory that contains the file `offlinepackager.exe`.
3. Enter the name of the tool along with options, if any, for example:

```
offlinepackager --input-file=sample.f4v --output-path=c:\sampleoutput
```

In this example, the following files are output:

- `sampleSeg1.f4f`
- `sampleSeg1.f4x`
- `sample.f4m`

On Linux

1. Open a terminal window.
2. At the shell prompt, enter the name of the tool along with options, if any, for example:

```
./offlinepackager --input-file=sample.flv --output-path=/sampleoutput
```

In this example, the following files are generated as output:

- sampleSeg1.f4f
- sampleSeg1.f4x
- sample.f4m

Packaging and encrypting content

Many command-line options are required to encrypt a file. It's easier to use a configuration file to set options than to set options at the command line.

1. Create a file offlinepackager_config.xml in a text editor.
2. Enter values for the following required options (sample values are included):

```
<offline>
  <input-file>someFile.f4v</input-file>
  <content-id>foo</content-id>
  <common-key>commonKey.bin</common-key>
  <license-server-url>
    http://myfaxserver.com
  </license-server-url>
  <license-server-cert>
    licenseServer.der
  </license-server-cert>
  <transport-cert>production_transport.der</transport-cert>
  <packager-credential>
    packagerCredential.pfx
  </packager-credential>
  <credential-pwd>mYpwd</credential-pwd>
  <policy-file>policyFile.pol</policy-file>
</offline>
```

You can pass additional parameters to the OfflinePackager. For more information, see ["Command-line options to fragment files"](#) and ["Command-line options to encrypt files"](#).

3. Save the configuration file.
4. Open a command window.
5. Navigate to the directory that contains the file offlinepackager.exe.
6. Execute the following:

```
offlinepackager --conf-file=offlinepackager_config.xml
```

Command-line options for fragmenting files

You can use the command line to set all the options for the OfflinePackager. Options set at the command line override any options you set in the configuration file. The only required option is `--input-file`.

Option	Description	Required	Default
<code>--input-file</code>	The path to the source container file. The path can be absolute or relative to the directory that contains the OfflinePackager. When specifying the <code>--conf-file</code> option, if the configuration file contains an <code><input-file></code> value, do not use the <code>--input-file</code> option.	Yes	None
<code>--output-path</code>	The path to the directory to which the packaged files should be written. The path can be absolute or relative to the directory containing the OfflinePackager.	No	Current directory
<code>--fragment-duration</code>	The target length of each fragment, in seconds. The actual length of each fragment depends on properties of the input file, such as the keyframe interval.	No	4
<code>--segment-duration</code>	The length of a segment in seconds. One segment will be created for the entire content if the value for segment duration is not specified or if the value is equal to 0.	No	0
<code>--conf-file</code>	The configuration file that contains settings for the packaging process. Specify an absolute path, or a path relative to the directory that contains the OfflinePackager tool. The tags in the configuration file correspond to these command line options.	No	None
<code>--simulate-live</code>	Simulates live stream packaging. This option can be used to simulate the live packaging scenario using the OfflinePackager tool.	No	None

Configure Protected HTTP Dynamic Streaming (PHDS)

You can use the HTTP Dynamic Streaming SDK Broadcast (HDKB) to generate protected content for Flash Player and Adobe AIR over HTTP without using a DRM License Server. When HDKB packages the content, it generates the license and embeds it into the DRM metadata of the content stream.

Configuration parameters required for PHDS to work are as follows:

```
<offline>
  <enable-phds>true</enable-phds>
  <input-file>sample.f4v</input-file>
  <common-key>sample-hdk-common-key</common-key>
  <content-id>sample-hdk-content-id</content-id>
  <license-server-url>http://myfaxserver.com</license-server-url>
  <policy-file>phds/static/phds_policy.pol</policy-file>
  <recipient-cert>phds/sd/ADBEXP.cer</recipient-cert>
  <license-server-cert>
    adobe-provided-license-cert.der
  </license-server-cert>
  <transport-cert>adobe-provided-transport-cert.der</transport-cert>
  <packager-credential>
    adobe-provided-packager-credential.pfx
  </packager-credential>
  <credential-pwd>
    adobe-provided-packager-credential-password
  </credential-pwd>
  <license-server-credential>
    adobe-provided-license-server-credential.pfx
  </license-server-credential>
  <license-server-credential-pwd>
    adobe-provided-license-server-credential-password
  </license-server-credential-pwd>
</offline>
```

For the list of PHDS configuration parameters, see the section on [“Command-line options to encrypt files”](#).

The recipient certificates and policy files required for PHDS are placed in the /phds directory.

The OfflinePackager sample takes as input a single recipient certificate. However one can provide multiple recipient certificates using the HDKB interface.

There is a set of two policy files placed at /phds/static directory. Details of the policy files are below,

Policy Name	Description
phds_24hr_policy.pol	24 Hour limited policy anonymous; 24 hours limited license caching. The 24 hours window starts when the DRM metadata is generated.
phds_policy.pol	Unlimited policy anonymous; not use license chaining; unlimited license caching; and binding to the Shared Domain is permitted This policy allows playback at any time.

The other certificates and credentials required for PHDS have to be obtained from Adobe.

The certificate enrollment guide outlines the process for requesting a certificate. http://www.adobe.com/go/learn_flashaccess_certificate_3 (Go to the section called "Requesting a certificate.")

Requesting PHDS certificates

To request PHDS certificates, generate the CSR file and email the file to FlashAccessCertRequest@adobe.com. Ensure that you include "PHDS" in the request, and provide three CSR files, one for each certificate type transport, license, and packager. The certificate would be provided within three business days.

For more information, contact the Adobe Access support team.

The OfflinePackager configuration file

The OfflinePackager uses an XML configuration file to read values for command-line options. It is often easier to specify options in a text file that you can reuse than to enter options at the command line.

Options set at the command line override options set in the configuration file. This feature lets you create a configuration file for a set of content, but edit a few settings for individual files.

Note: Using the configuration is not required to run the tool.

Enter the `--conf-file` option at the command line to pass the name and location of a configuration file.

You can enter all the options for packaging and encrypting, including the input file, in the configuration file.

The configuration file has the following format:

```
<offline>
  <input-file>/media/input-file.f4v</input-file>
  <output-path>/media/http</output-path>
  <fragment-duration></fragment-duration>
```

```
</offline>
```

Note: Set as many options as you want from the list of supported tags. For options that accept a path, specify absolute paths or paths relative to the directory that contains the tool.

Command-line options for encrypting files

Use OfflinePackager to generate protected content for Flash Player and Adobe AIR over HTTP. Pass digital certificates for some of the following options. To obtain digital certificates, see the [Adobe Access Certificate Enrollment Site](#).

Option	Description	Required	Default
--transport-cert	The path to a DER encoded transport certificate file.	Yes	None
--license-server-url	URL of the license server that handles license acquisition for this content.	Yes	None
--license-server-cert	DER encoded license server certificate file used to verify license server authenticity.	Yes	None
--packager-credential	The path to a PFX file containing the packager's protection credentials.	Yes	None
--credential-pwd	Password string used to secure packager credentials.	Yes	None
--common-key	File containing the base key. The base key is used with the content ID to generate the content encryption key. The base key must be randomly generated and must be 16 bytes in length. For dynamic streaming (also called multi-bitrate streaming), use the same common key and content ID for an entire content set. Using the same key and ID allows a single license to decrypt a set of content. Common key should be randomly generated using a high quality random number generator. Note: If you are using the content encryption key, the common key is not required.	No	None
--content-id	Unique identifier of the media. Also used with the common key to generate the content encryption	Yes	None

	key.		
--content-encryption-key	128-bit content encryption key used to encrypt the file. Caution: The common key and the content encryption key cannot be used simultaneously.	No	None
--policy-file	File containing the policy for content. Currently, only a single policy can be applied to content packaged with this tool.	Yes	None
--encrypt-audio	Specifies whether to encrypt the audio in the file.	No	True
--encrypt-video	Specifies whether to encrypt the video in the file.	No	True
--encrypt-data	Specifies whether to encrypt the data in the file. This option supports both FLV and F4V/MP4 file formats.	No	True
--video-encrypt-level	The level of encryption for H.264 content. Possible values are 0 (low), 1(medium), and 2 (high). The default value is 2. Using this option on non-H.264 content doesn't generate an error. The values "0" and "1" mean "partial encryption". Only samples like video keyframes are encrypted. This setting creates fewer frames to decrypt. Use this setting to improve playback performance on the client. The value "2", encrypts all video samples (after --ms-unencrypted, if specified). See, Adobe Access Documentation for partial encryption for more details	No	2
--enable-phds	Specifies whether to enable the Protected HTTP Dynamic Streaming (PHDS) feature.	No	false
--license-server-credential	The path to a PFX file containing the license server's protection credentials. The path can be absolute or relative.	Yes (When PHDS is enabled)	None

<code>--license-server-credential-pwd</code>	Password string used to secure the license server credentials.	Yes (When PHDS is enabled)	None
<code>--recipient-cert</code>	DER encoded recipient certificate file. The file path can be absolute or relative to the directory that contains the OfflinePackager. The OfflinePackager can use multiple certificates.	Yes (When PHDS is enabled)	None
<code>--additional-license</code>	The additional license file. The file path can be absolute or relative to the directory that contains the OfflinePackager. The file must contain any additional licenses to be embedded into an additional DRM header.	Yes (When PHDS is enabled)	None
<code>--encrypt-keys-rotation-enable</code>	Switches key rotation on/off. See the Key Rotation section in http://www.adobe.com/go/learn_flashaccess_protectcontent_3	No	false
<code>--encrypt-keys-rotation-interval</code>	Interval (in seconds) after which the content encryption key is changed.	No	900
<code>--encrypt-keys-rotation-file</code>	Contains the rotation keys. This file contains a sequence of rotated keys used to encrypt content when key rotation is enabled. If no file is specified, a key is randomly generated with the sample. This key is used for the key rotation interval specified and then a new key is generated for the next interval, and so on. The keys must be 16 bytes in length and specify keys as hexadecimal values (as shown below). Whitespaces between the hexadecimal values are optional. When multiple keys are specified, the keys are cycled through in the order they are specified. The file must contain the keys in the following format: 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20	No	None
<code>--generate-chained-license</code>	When <code>--generate-chained-license</code> value set to true, a	No	false

	<p>chained licence is generated. To stop generating chained licence, set value of the <code>--generate-chained-license</code> directive to false.</p> <p>Note: To use this option, enable PHDS along with all its configuration parameters except DER-encoded recipient certificates.</p>	Requires a policy with Enhanced License Chaining enabled. Enhanced License Chaining is described in http://www.adobe.com/go/learn_flash_access_protectcontent_3	
<code>content-encryption-key-id</code>	<p>This parameter is used to enable external CEK feature and to provide CEK-ID associated with CEK. This parameter can't be specified with the <code>--common-key</code> parameter. Instead you should specify the <code>--content-encryption-key</code> parameter.</p>	No	None

Command-line options for license rotation

Option	Description	Required	Default
<code>--enable-license-rotation</code>	Enables license rotation.	No	false
<code>--license-rotation-interval</code>	The interval in hours after which to rotate the license information. You can provide the value in fractions. For example, 1.32.	No	None
<code>--license-info-file</code>	This file contains additional information required for license generation. You can specify additional parameter for the AAXS meta-data information that is to be generated for individual license	No	None

Command-line options for blackout

Option	Description	Required	Default
<code>--enable-blackout</code>	Specifies whether to enable blackout	No	false
<code>--scte35-mode</code>	Splice Cue insertion mode. If true then all the SCTE-35 messages extracted from input file will be inserted in scte35 mode else in		

	simple mode, it is required when blackout is enabled		
--	--	--	--

The OfflinePackager design

The following section details the source code design of the Offline Packager sample application.

The `PackagerConfig` structure defines the mandatory parameters for the `OfflinePackager`. Values from the command line or the config file are used to build this structure. This structure is passed as a parameter to the `OfflinePackager::fragmentFile` call.

The `OfflinePackager::fragmentFile` call accepts the mandatory values from the structure passed to it, builds the `MediaSource` class, and initializes the Packager instance.

`MediaSource` reads input files and returns media samples from the input file. An iterator on `MediaSource` retrieves the media samples from the input. These media samples are pushed to the Packager using the `IPackager::pushMediaMessage` call. When all the media samples that need to be fragmented are supplied to the Packager, `IPackager::notifyEndOfStream()` is called to conclude the stream. Then, the packager finishes fragmenting the input media messages into F4F assets, and creates the .f4m file for the F4F assets.

The `FlvMediaSource` class implements `IMediaSource`, which provides utility functions to traverse the complete media (FLV) file and obtain the duration of the input media file. See the definition of `FlvMediaSource::getSampleIterator` and `FlvMediaSource::getDuration` for details. The `FLVMessageIterator` class also implements `IMessageIterator`, assists in pulling out media messages from the input file, and checks the availability of additional media messages. See the definition of `getNextMsg` and `hasMoreEntries` for details. To understand how to create a `MediaMessage` stream to be passed as input to the `Fragmenter`, see the implementation of the class.

Here is a sample code for the packager design:

```
<offline>
  <input-file>../resources/input/scte-
35_w_forced_i_frames_1500_768x432_1900_4secGOP.ts</input-file>
  <output-path>../resources/output/output_long_file</output-path>
  <duration-precision>0.001</duration-precision>
  <fragment-duration>4</fragment-duration>
  <segment-duration>40</segment-duration>
  <common-key>../resources/input/dme/common-key.bin</common-key>
  <content-id>ContentId1</content-id>
  <license-server-url>http://xxxxxxx</license-server-url>
  <license-server-cert>../resources/input/dme/QE-DRMTEST-EVAL-PRO-
20120830.der</license-server-cert>
  <transport-cert>../resources/input/dme/QE-DRMTEST-EVAL-PRO-
20120830.der</transport-cert>
  <packager-credential>../resources/input/dme/QE-DRMTEST-EVAL-PRO-
20120830.pfx</packager-credential>
```

```

<credential-pwd>kY2IUPnQuG0=</credential-pwd>
<policy-file>../resources/input/dme/policy01_anonymous.pol</policy-file>
<enable-license-rotation>true</enable-license-rotation>
<license-rotation-interval>0.0083333333333333</license-rotation-
interval>
  <license-info-file>licenseFile-4.xml</license-info-file>
</offline>

```

Packager design

The `Packager` class acts as middleware, with applications, such as the `OfflinePackager`, forming the top layer. The SDK forms the lower layer, which fragments input media messages.

The `Packager` class is the sample implementation of the process of converting a stream of media messages to F4F assets. The class can perform encryption and fragment media messages simultaneously. This class uses `Fragmenter`, `FragmentDeterminer` and `FragmentConsumer`, and `F4XWriter` to do its job. The `Packager` class can be used in both the offline use case (for example the `OfflinePackager`) and the live use case. The class only requires the user application to supply properly formed media messages.

To use the `Packager`, initialize the `Packager` by calling `IPackager::initializePackager` with the `PackagerConfig` structure, a boolean value indicating whether the `Packager`'s input stream is a live stream or a file stream (offline), and an object ID used for logging purposes. The initialization process creates objects of the classes `Fragmenter`, `FragmentConsumer`, `FragmentDeterminer` and `F4XWriter`. `MoovBuffer`, which is fetched from the `FragmentConsumer` is set on the `Fragmenter` before proceeding with fragmentation.

The `FragmentDeterminer` utility class determines when to create fragments. `FragmentDeterminer` is a state machine that processes stream messages and determines the points where fragments start and end. It also determines an appropriate time interval to introduce gaps in the `Fragment Runtable`. Metadata related to each message (for example: timestamp, audio, video, or data message; keyframe) is passed to the `fragmentDeterminer::fragment()` method. This method returns `FragmentResult`, a `std::pair` object consisting of `FragmentStatus` and `FragmentBoundaryInfo`. The algorithm ensures that the number of fragments is predictable, irrespective of where the streaming starts. It helps to maintain consistency across redundant fragmentation systems even in cases where some of them start fragmentation at different points.

The `FragmentConsumer` class consumes fragments that are produced by the `Fragmenter`. `FragmentConsumer` provides the `Fragmenter` a reference to its `MoovBuffer`, which is updated by the `Fragmenter` when a new fragment is generated. The fragments created by the `Fragmenter` are provided to the `FragmentConsumer` through `pushFragment()`. `FragmentConsumer` writes this fragment to an F4F file. Along with the fragment, the fragment context is also passed to the `FragmentConsumer` through `pushFragment()`, which is used to write F4F files. When a gap is introduced in the fragment run table using an `addSkip` call to the `fragmenter`, the corresponding skip is added to the `FragmentConsumer` using the `IFragmentConsumer::skipFragment()` call. If there

are multiple segments, a new `FragmentConsumer` is set for every segment, and its `MoovBuffer` is reassigned to the `Fragmenter`.

An `F4XWriter` is used to generate F4X files. `F4XWriter` is passed as a parameter to initialize `FragmentConsumer`, which also writes F4X files. `F4XWriter` must be created before setting up the `FragmentConsumer`. Every new `fragmentConsumer` object is passed the same `F4XWriter` object, which creates multiple F4X files in the case of multiple segments.

After the `Packager` is initialized, the application can push media messages into the `Packager` using the `IPackager::pushMediaMessage` function. Here the `FragmentDeterminer` determines the action to be taken when the current message arrives. The `FragmentDeterminer` may guide the packager to push the message into the pending message queue. Alternatively, the `Packager` requests that the `Fragmenter` creates the fragment from the messages in the message queue, using `IFragmenter::pushMessage`, `IFragmenter::completeFragment` (to complete the fragment), and `IFragmenter::completeSegment` (if the current segment duration is greater than or equal to the user specified segment duration). After completing a segment, a new `FragmentConsumer` is created to receive fragments of the next segment and write them to a different F4F file. In the live use case, when the first fragment is created, the meta file containing the metadata information of the stream is created. In addition, the `Packager::writeF4M` function is used to create the F4M file, with the bootstrap information, meta, and drmmeta files. In the live use case, the F4M file is updated periodically on the creation of new fragments to include the latest bootstrap information.

To stop the fragmentation process, the application calls the `IPackager::notifyEndOfStream` function. This function ensures that the `Packager` completes the fragmentation process. Residual messages in the pending message queue are pushed into the `Fragmenter`, and the resulting fragment is pushed into `FragmentConsumer`. For the live use case, the bootstrap file is updated with the latest bootstrap information. The `Fragmenter` is notified of the end of stream by calling `IFragmenter::notifyEndofStream`. In the offline use case, the F4M file is written using `Packager::writeF4m` function, at the end of the fragmentation process.

License Rotation

The License Rotation feature allows you to change the license information of HDS content at regular intervals while the content is packaged. The license information here refers to Content Encryption Key (CEK) or usage policies or both.

If License Rotation is enabled, more than one metadata is inserted per stream in the manifest file (.f4m). In this case, an HDS client acquires more than one license at regular intervals. It needs to decrypt the license at regular intervals to play the packaged HDS content.

The following two new members have been introduced in the `PackagerConfig` structure to configure license rotation.

- `PackagerConfig::m_enableLicenseRotation`: This parameter is used to enable license rotation.

- `PackagerConfig::m_licenseRotationInterval`: This parameter used to provide interval after which license should be rotated. Value in milliseconds.
- `PackagerConfig::m_licenseInfoFile`: This parameter is used to specify a license information file.

A new interface, `ILicenseInfoProvider`, is introduced. This interface is used by the instance of *IPackager* to acquire the next license information, using

`ILicenseInfoProvider::getNextLicenseInfo()` API. A reference implementation of this interface is provided within the Packager module. A custom implementation for the same can be created using the newly introduced `IPackager::setLicenseInfoProvider()` API. The default implementation of this interface generates new license information by random modifications in the common encryption key (CEK). It uses `PackagerConfig::m_fragmenterConfig::m_EncryptionConfig` configuration parameters while initializing the *IPackager*.

```

/// \namespace adbe
namespace adbe
{
    /// \namespace hdssamples
    namespace hdssamples
    {
        /// \interface ILicenseInfoProvider
        /// \brief An interface to be used by packager to get new LicenseInfo
        ///        for License Rotation.
        class ILicenseInfoProvider
        {
        public:
            /// Constructor
            /// \param [in] objId Object Id for logging
            ILicenseInfoProvider(const fms::String& objId = ""):
            m_ObjId(objId) {}

            /// Pure virtual function to get new license information needed
            /// to generated new license and encrypt contents.
            /// \param [out] licenseInfo License information related to new
            /// license
            /// \return Success or error code indicating the error.
            virtual adbe::hds::ErrorCode
            getNextLicenseInfo(adbe::hds::LicenseInfo& licenseInfo) = 0;

            /// Virtual function to initialize ILicenseInfoProvider.
            /// \return Success or error code indicating the error.
            virtual adbe::hds::ErrorCode initialize() { return
            adbe::hds::eSuccess; };

            /// Virtual function to finalize ILicenseInfoProvider.
            /// \return Success or error code indicating the error.
            virtual adbe::hds::ErrorCode finalize() { return
            adbe::hds::eSuccess; };
    }
}

```

```

        /// virtual destructor
        virtual ~ILicenseInfoProvider() {}
    protected:
        /// Object id for logging.
        fms::String m_ObjId;
    };
} //namespace hdssamples
} //namespace adbe

```

The following API is used to provide custom implementation of *ILicenseInfoProvider* to *IPackager*.

```

{
namespace adbe
{
    namespace hdssamples
    {
        class IPackager
        {
            ...

            /// Function to set custom implementation of
            ILicenseInfoProvider.
            /// \param [in] licenseInfoProvider Shared pointer of an
            instance of custom implemenation of ILicenseInfoProvider.
            /// \return Success or error code indicating the error.
            ErrorCode setLicenseInfoProvider (boost::shared_ptr<
            ILicenseInfoProvider > licenseInfoProvider)

            ...
        };
    }
}

```

Changes in HDS manifest for license rotation (New in 3.2)

The manifest file holds license rotation information. The following changes are made to the file based on the new process by which the client would be triggered to acquire new license for upcoming encrypted fragments.

New element <drmAdditionalHeaderSet>

A new optional element <drmAdditionalHeaderSet> introduced. This tag will be used to hold and collectively refer all the <drmAddtionalHeader> elements that are associated with a particular stream (<media> element) of manifest.

- This element may be present (It is required only in case of License Rotation).
- This element shall not be present in set-level manifest.

- More than one <drmAdditionalHeaderSet> elements may be present in case of single-level manifest.
- More than one <drmAdditionalHeaderSet> elements shall not be present in case of stream-level manifest.
- This element must be child of root (i.e. <manifest>) element.
- This element can only contain one or more than one <drmAdditionalHeader> element as its children.
- This element's scope shall be the file it resides in. A stream-level manifest's <media> element shall not reference a <drmAdditionalHeaderSet> element from a different stream-level manifest. Therefore, there is no conflict when two <drmAdditionalHeaderSet> elements from different files have the same id.

Attributes of tag

@id: shall be the ID of this element. This attribute may be present. When this attribute is not present then this <drmAdditionalHeaderSet> element shall apply to all <media> elements that don't have a @drmAdditionalHeaderSetId attribute. When this attribute is present then this <drmAdditionalHeaderSet> element shall apply only to those <media> elements that have the same ID in their @drmAdditionalHeaderSetId attribute.

Modification in <drmAdditionalHeader> element

The <drmAdditionalHeader> element is modified. A new optional attribute @startTimestamp may be present. This specifies the timestamp of the very first message (of the associated stream(s)) from which the DRM Meta data contained in this <drmAdditionalHeader> element applies. If this attribute is not present then it is assumed to have a value same as that of the timestamp of first message (as per the current bootstrap) of the associated stream(s). Units for this element shall be fractional seconds.

Modification in <media> element

The <media> element is also modified. A new optional attribute @drmAdditionalHeaderSetId may be present. This attribute associates the <media> elements to a <drmAdditionalHeaderSet> whose @id attribute is having the same value as that of this attribute. When this attribute is not present then this <media> element shall be associated with the <drmAdditionalHeaderSet> element which is not having the @id attribute.

The following is an example for a manifest file that contains license rotation information:

```
<manifest>
  <media streamId="livestream" url="..." bootstrapInfoId="bootstrap794"
  drmAdditionalHeaderSetId="drmMetadataSet203"/>
  <media streamId="livestream1" url="..." bootstrapInfoId="bootstrap562"
  drmAdditionalHeaderSetId="drmMetadataSet404"/>

  <bootstrapInfo profile="named" id="bootstrap794" >XXXXXX</
bootstrapInfo >
  <bootstrapInfo profile="named" id="bootstrap562" >XXXXXX</
bootstrapInfo >
```

```

    <drmAdditionalHeaderSet id="drmMetadataSet203">
      <drmAdditionalHeader id="drmMetadata7807"
startTimestamp="0">XXXXXX </drmAdditionalHeader>
      <drmAdditionalHeader id="drmMetadata7808" startTimestamp="45236">
XXXXXX </drmAdditionalHeader>
    </drmAdditionalHeaderSet>

    <drmAdditionalHeaderSet id="drmMetadataSet404">
      <drmAdditionalHeader id="drmMetadata7555"
startTimestamp="0">XXXXXX </drmAdditionalHeader>
      <drmAdditionalHeader id="drmMetadata8677" startTimestamp="55236">
XXXXXX </drmAdditionalHeader>
    </drmAdditionalHeaderSet>

</manifest>

```

Regional blackout

To support regional blackouts, the attributes of <cue> elements of HDS manifest file has been modified as below:

- @cue: (Optional) It is base-64 encoded string of SCTE35 packet.
- @type: It has two values: SpliceOut (Simple Mode) and scte35 (SCTE35 Mode).
- @id: It refers to the signal id. For SpliceInsert command, the value is splice_event_id. For time_signal, it is segmentation_event_id.
- @time: It refers to the time adjusted by pts_adjustment in splice_info_section. For the SpliceInsert command, the value is splice_time.pts_time. For time_signal, it is pts_time.
- @duration: (Optional) It describes the duration of the event. For SpliceInsert, break_duration.duration is used. For Time_Signal command, the segmentation_duration attribute is used.
- @availNum: (Optional) As per SCTE-35 spec. If zero, the tag is ignored.
- @availsExpected: (Optional) As per SCTE-35 spec. If zero, the tag is ignored.

The following is an example for a manifest file that contains blackout information:

```

<manifest>
  <media streamId="livestream" cueInfoId="cueInfoId_1" url="..."
bootstrapInfoId="bootstrap794"
  drmAdditionalHeaderSetId="drmMetadataSet203" />
  <bootstrapInfo profile="named" id="bootstrap794" >XXXXXX</
bootstrapInfo>

  <drmAdditionalHeaderSet id="drmMetadataSet203">
    <drmAdditionalHeader drmContentId="..." id="drmMetadata7807"
startTimestamp="0">XXXXXX </drmAdditionalHeader>
    <drmAdditionalHeader drmContentId="..." id="drmMetadata7808"
startTimestamp="51.767">XXXXXX </drmAdditionalHeader>
  </drmAdditionalHeaderSet>

```

```

<drmAdditionalHeader drmContentId="..." id="drmMetadata7809"
startTimestamp="61.767"> XXXXXX </drmAdditionalHeader>
<drmAdditionalHeader drmContentId="..." id="drmMetadata7810"
startTimestamp="112.441">XXXXXX </drmAdditionalHeader>
<drmAdditionalHeader drmContentId="..." id="drmMetadata7811"
startTimestamp="122.449"> XXXXXX </drmAdditionalHeader>
</drmAdditionalHeaderSet>

<cueInfo id="cueInfoId_1">
<cue type="scte35" id="1" time="51.767" duration="10.000"
programId="2345" availNum="1" availsExpected="5"
cue="/DBsAAAAAAAA/wAQFAUAAAAFf+/" />
<cue type="scte35" id="1" time="61.767" duration="0"
cue="/DBsAAAAAAAA/wAQFAUAAAAFf+/" />
<cue type="scte35" id="1073741858" time="112.434" duration="10.000"
cue="/DBsAAAAAAAA/WBQbwAQFAUAAAAFf+/" />
<cue type="scte35" id="1073741858" time="122.434" duration="0"
cue="/DBsAAAAAAAA/wAQFAUAAAAFf+/" />
</cueInfo>
</manifest>

```

Trick Play support in HDKB

The OfflinePackager sample application has been updated to generate an F4M file that instructs the Adobe Primetime player that a keyframe only stream corresponding to the actual stream can be delivered by the web server. This stream can be used by the Adobe Primetime player to perform Trick Play.

Note: For more information on Trick Play, see the *HDKB 3.1 Overview document*.

In HDKB 3.0, a new parameter `--add-keyframe-only-media` has been added to the list of allowed parameters for OfflinePackager.

When you use this parameter, the OfflinePackager adds a `<media>` element with type `video-keyframe-only` in the F4M file. The presence of this `<media>` element instructs the player that the web-module serving the stream is capable of providing a stream containing video key frames only. Such a stream can be used by the player to perform fast forward or fast rewind trick plays.

The following example shows the new key frame only `<media>` element:

```

<media url="vod/kfonly/stream1" type="video-keyframe-only" bitrate="800"
/>

```

The `Packager::writeF4M` function and the `F4MWriter` class have been updated to generate the new key frame only media elements.

Note: The F4M specification has also been updated to include a new `@version` attribute for the `<manifest>` element. The absence of the `@version` attribute is equivalent to setting the

@version to 1.0 for backward compatibility. For trick play to work, the Adobe Primetime player requires @version to be set to 3.0

The following example shows a manifest file (F4M) with the new key frame only <media> element:

```
<manifest xmlns="http://ns.adobe.com/f4m/1.0" version="3.0">

    <bootstrapInfo profile="named" id="bootstrap2594">... </bootstrapInfo>
    <media url="vod/stream" bitrate="2200"
        bootstrapInfoId="bootstrap2594" />

    <!-- The Keyframe only media elements with their new URLs -->
    <media url="vod/kfonly/stream" type="video-keyframe-only"
        bitrate="2200" />
</manifest>
```

The presence of “kfonly/” keyword just a level before the actual stream name in the HTTP request, instructs the web module to serve the key frame only fragment to the player. The web module reads the actual fragment from the disk (the path of the actual fragment can be found by removing “kfonly/” from the request URL before mapping the URL to the disk) and then creates the key frame only fragment from the actual fragment just in time.

Similarly, you can also use the set-level and stream-level manifests (F4M 2.0) system to instruct the player of the origin web module’s (mod_f4http’s) ability to deliver key frame only fragments. The following example shows a set-level manifest file with the video-keyframe-only <media> element (make sure to set the @version attribute to 2.0 as per new F4M specifications while creating Set-Level Manifests):

```
<manifest xmlns="http://ns.adobe.com/f4m/1.0" version="2.0">
    <media href="stream.f4m" bitrate="2200" />
    <media href="stream-KFOnly.f4m" type="video-keyframe-only"
        bitrate="2200" />
</manifest>
```

External CEK support in HDKB (New in 3.5)

In HDKB 3.5, the OfflinePackager sample application is updated to support the External CEK feature.

However, the sample packager/offlinepackager module does not provide a sample code for implementing a shared secret recipe to use the feature. This is left to the user.

To demonstrate the basic implementation of the feature, you must enter values for both CEK_ID and CEK as a pair through application parameters.

You can specify the CEK_ID using the parameter `--content-encryption-key-id`. Use the parameter `--content-encryption-key` to specify the associated CEK.

Note: You cannot use the `--common-key` parameter with the External CEK feature.

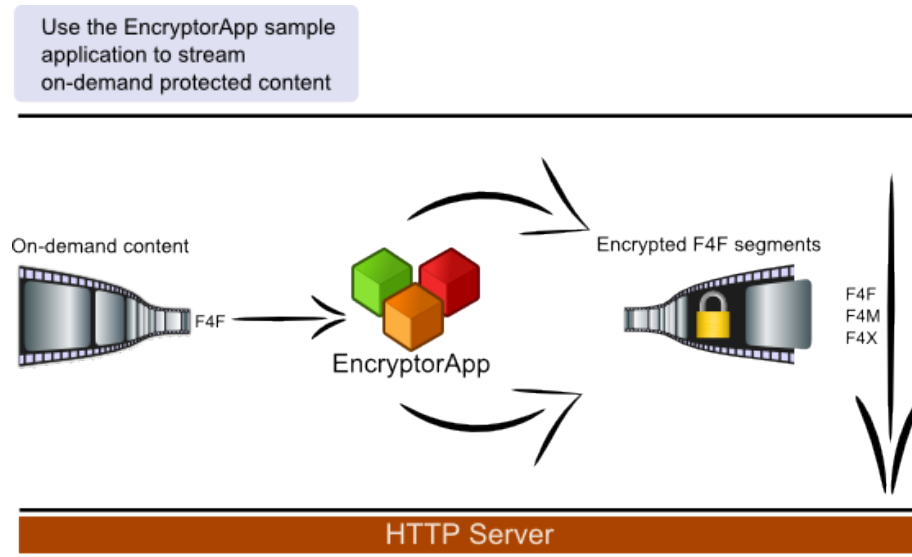
Sample modules restrict using this feature with the license rotation or blackout feature. To use these features simultaneously, implement them using the HDKb core API

```
IFragmenter::updateLicense().
```

EncryptorApp

The EncryptorApp is a command line tool that accepts a pre-packaged input file or a media base name as input and creates an encrypted fragment file. The media base name also includes the name of the stream. The tool looks for f4f and f4x files, with the media base name as the prefix.

Figure 2: Use EncryptorApp to stream on-demand protected content



The command-line options available for encryption in the EncryptorApp are identical to the OfflinePackager.

Tip: Use a configuration file to set options.

In the following example, the `EncryptorApp` looks for all F4F and F4X files that have *sample* as a prefix (for example, `sampleSeg1.f4f`, `sampleSeg1.f4x`, and `sampleSeg2.f4f`) in the `voddir/output` directory. It iterates through multiple segment files, if they exist.

```
<offline>
  <media-base-name>voddir/output/sample</media-base-name>
  <content-id>contentId</content-id>
  <common-key>commonKey.bin</common-key>
  <license-server-url>
    http://myfaxserver.com
  </license-server-url>
  <license-server-cert>licenseServer.der</license-server-cert>
  <transport-cert>transportCert.der</transport-cert>
  <packager-credential>packagerCredential.pfx</packager-credential>
  <credential-pwd>mYpwd</credential-pwd>
  <policy-file>policyFile.pol</policy-file>
</offline>
```

Note: Add optional elements to the configuration file to use non-default values for those elements.

EncryptorApp design

The `EncryptorAppConfig` structure defines mandatory parameters for the `EncryptorApp`. The values from the command line or the config file are used to build this structure. The structure is passed as a parameter to `EncryptorApp::EncryptFile()`. `EncryptorApp::EncryptFile` accepts the required values from the structure passed to it and builds the `FragmentExtractor`, `Encryptor` and `FragmentConsumer` classes. The class `MoovBuffer`, fetched from the `FragmentConsumer`, is set on the `Encryptor` before proceeding with encryption. `FragmentExtractor` reads input files (F4F in this case) and returns fragments from the input file. An iterator on the `FragmentExtractor` is used to retrieve the fragments and then passed to the `Encryptor` using `Encryptor::encryptFragment()`.

`Encryptor::encryptFragment` updates the `FragmentInfo` passed to it by reference.

`FragmentInfo` holds the encrypted fragment and any contextual information of the encrypted fragment.

`Encryptor::notifyEndOfStream()` is called to notify the end of stream on the encryptor. The last encrypted fragment information is used to write an F4M file using the `Encryptor::writeF4M` function.

The `FragmentConsumer` class consumes fragments produced by the `Encryptor`. `FragmentConsumer` references its `MoovBuffer` to the `Encryptor`, which is updated by the `Encryptor` when a new fragment is generated. The fragments created by the `Encryptor` are passed to the

`FragmentConsumer` through `pushFragment()`. Along with the fragment, the fragment context is also passed to the `FragmentConsumer` through `pushFragment`, which is used to write F4F files.

An `F4XWriter` is used to generate F4X files. `F4XWriter` is passed as a parameter for initializing `FragmentConsumer`, which writes F4X files.

MPEGParser

Use the `MPEGParser` to convert MPEG-2 TS data into F4F fragments using `IMpegParser` and `IMpegParserDelegate` interfaces. The `IMpegParser` interface parses MPEG-2 TS byte buffer to produce corresponding H.264/AAC/MP3 PES packets. On completing the packets, `IMpegParser` interface transfers completed H.264/AAC/MP3 PES packets to `IMpegParserDelegate` interface. `IMpegParserDelegate` interface converts PES packet into an HDKB consumable `MediaMessage` and pushes the `MediaMessage` to the packager. The packager bundles these messages into an F4F file.

The declaration of these two interfaces is as follows:

IMpegParser

```
/// \interface IMpegParser
/// \brief Represents the mpeg parser, which parses the MPEG TS data
class IMpegParser
{
public:
    /// Called to create an instance of IMpegParser
    /// \param[out] mpegParser provides a handle to IMpegParser
    /// instance created
    /// \param[in] pParserDelegate A handle to IMpegParserDelegate
    /// \param[in] pHandler A handle to IWarningHandler
    /// \param[in] objectId objectID that will be used while
    /// creating logs
    /// \return Success or error code indicating the error.

    static ErrorCode create(IMpegParser** mpegParser,
        IMpegParserDelegate* pParserDelegate,
        IWarningHandler* pHandler,
        const fms::String& objectId = "");

    /// Called to destroy the IMpegParser instance
    /// \param[in] mpegParser IMpegParser instance to be destroyed
    /// \return Success or error code indicating the error.

    static ErrorCode destroy(IMpegParser** mpegParser);

    /// Called to pass in MpegTS data into IMpegParser for parser
    /// \param[in] buf ByteBuffer containing MpegTS data
    /// \param[in] size size of the data in the ByteBuffer
    /// \return Success or error code indicating the error.

    virtual ErrorCode appendBytes(ByteBuffer& buf, uint32_t size)
    = 0;
```

```

    /// Called to tell the parser to flush out any data
    queued and close.
    /// \param bFlush If true, it tells the mpeg parser to flush any
    /// queued data to IMpegParserDelegate.
    /// \return Success or error code indicating the error.

    virtual ErrorCode closeParser(bool bFlush) = 0;

    /// IMpegParser Destructor
    virtual ~IMpegParser()
    {
    }

};

```

IMpegParserDelegate

```

enum Codec
{
    kCodecH264 = 0x01,
    kCodecAAC,
    kCodecMp3,
    kCodecUnsupported = 0xFF
};

enum SpliceCommand
{
    kSpliceNull = 0x01,
    kSpliceSchedule = 0x04,
    kSpliceInsert = 0x05,
    kSpliceTimeSignal = 0x06,
    kSpliceBandwidthReservation = 0x07,
    kSplicePrivateCommand = 0xFF,
    kSpliceInvalid
};

/// \interface IMpegParserDelegate
/// \brief Represents the delegate object to which IMpegParser passes
/// PES data

class IMpegParserDelegate

```

```

{
public:

    /// Called by IMpegParser to send out the complete PES packet
    /// \param[in] packetID PID of the the packet on which this
    /// data is coming in the TS stream
    /// \param[in] codecType codec of the PES packet
    /// \param[in] pts Presentation timestamp of the PES packet
    /// \param[in] dts Decoding timestamp of the PES packet
    /// \param[in] buf ByteBuffer containing the actual PES payload
    /// \param[in] readIdx Index from which ByteBuffer should be read
    /// \param[in] length size of the PES payload
    /// \return Success or error code indicating the error.

    virtual adbe::hds::ErrorCode onCompletePkt(
        uint32_t packetID, Codec codecType, uint64_t pts,
        uint64_t dts, ByteBuffer& buf, uint32_t readIdx,
        uint32_t length) = 0;

    /// Called by IMpegParser to send out the complete
    /// but unsupported
    /// PES packet whose codec is not supported
    /// \param[in] packetID PID of the the packet on which this
    /// data is coming in the TS stream
    /// \param[in] buf ByteBuffer containing the actual PES payload
    /// \param[in] readIdx Index from which ByteBuffer should be read
    /// \param[in] length size of the PES payload
    /// \return Success or error code indicating the error.

    virtual adbe::hds::ErrorCode onTSNotSuppPacket(
        uint32_t packetID, ByteBuffer& buf, uint32_t readIdx,
        uint32_t length) = 0;

    /// Called by IMpegParser to flush the leftover data
    /// \return Success or error code indicating the error.

    virtual adbe::hds::ErrorCode flushData() = 0;

    /// Called by IMpegParser to send out the complete SCTE packet
    /// \param[in] packetID PID of the the packet on which
    /// this data is coming in the TS stream
    /// \param[in] buf ByteBuffer containing the actual
    /// splice payload
    /// \param[in] readIdx Index from which ByteBuffer should be read
    /// \param[in] length size of the splice payload
    /// \param[in] ptsAdjustment The offset to be added to
    /// the splice's presentation time
    /// \return Success or error code indicating the error.

    virtual adbe::hds::ErrorCode onSCTESplicePkt(uint32_t
        packetID, ByteBuffer& buf, uint32_t readIdx,
        uint32_t length, uint64_t ptsAdjustment) = 0;

};

```

Considerations for using MpegParser

Ensure that following considerations are taken into account before using the sample code to convert MPEG-2 TS data to F4Fragments:

- For video, only H.264 codec is supported.
- For audio, mp3 and AAC codecs are supported.
- TS contents including multiple Program Streams are not supported.
- Ensure that MPEG-2 TS file has .ts extension.
- From HDKB 3.0, if the TS content has multiple audio and video elementary streams, only one of each type is packaged (whichever stream invokes the `IPackagerDelegate` first through the `onCompletePkt` API).

MPEGParser - Support for Closed Captioning (New in 3.0)

MPEGParser can parse the MPEG2TS stream and can package the audio, video and data messages in HDS format. HDKB handles the captioning data through the following workflow:

1. MPEGParser extracts the 608/708 closed captioning data from H264 SEI NALUs of type SEI_USER_DATA_REGISTERED_ITU_T_35.
2. MPEGParser creates the `onCaptionInfo` message and Packager multiplexes it before passing it to HDKB.

The `onCaptionInfo` message

The `onCaptionInfo` message is an AMF0 encoded data message containing an AMF0 encoded object. The AMF0 encoded object contains the name and value pairs.

The `onCaptionInfo` must contain the following parameters:

Parameter	Description
<code>onCaptionInfo.type</code>	<p>The <code>onCaptionInfo.type</code> indicates the caption type. The value must be "708".</p> <div> Note: The 608 captioning data that is sent as 708 data must also specify the type as "708" </div>
<code>onCaptionInfo.data</code>	<p>The <code>onCaptionInfo.data</code> provides the caption data. The value must be the Base64 encoding of one or more length-prefixed caption data chunks.</p> <p>The length-prefix is a 4-byte integer in network byte order expressing the length (in bytes) of the binary representation of the caption data chunk.</p>

Each caption data chunk is a binary blob that conforms to the definition of `cc_data` as specified in [CEA-708-D 2008](#), with the following exceptions:

- The third bit (zero bit) of `cc_data` may be either 1 or 0.
- Extra trailing bytes may be present beyond the final byte of the `cc_data`. These extra bytes, if included, must be accounted for in the length prefix.

Note: See the bundled `onCaptionInfo` specification document for more information.

Generating `onCaptionInfo` messages

The *Offlinepackager* sample application can be used to generate and push the `onCaptionInfo` data messages to the packager when an MPEG-TS file with caption data in H.264 SEI NAL unit is given as an input. In the input file, the captioning data should be embedded within the H.264 SEI NALU as specified in [ANSI/SCTE 128 2010 Section 8](#).

API changes

A new API, `pushCaptionDataMessage` has been introduced to enable pushing of AMF0 encoded captioning data messages to the packager. The `pushCaptionDataMessage` takes a copy of the media message to be pushed along with the decoding timestamp of the media message as arguments and returns the success or error code indicating the status of the action.

Note: The sample code in the *pestomediainfo* project depicts how captioning data can be extracted from a SEI NAL unit and how an `onCaptionInfo` data message can be generated and pushed to the packager.

A new class, `CaptionExtractor` has been introduced to extract captioning data from a SEI NAL unit. The function `CaptionExtractor::extractCaptionMsgFromH264Pkt` is used to extract data to create a media message. The media message can then be pushed to the packager using the `pushCaptionDataMessage` API.

The following code shows the `CaptionExtractor` class:

```
class CaptionExtractor
{
public:
    /// Default constructor
    CaptionExtractor( adbe::hds::IMemoryArena* pMemArena,
                    CCTimestampStrategy tsStrategy = ePresentationTime);

    /// Destructor
    ~CaptionExtractor();

    /// This method overrides the NALLength field size
    /// that reflects the size of the field prefixed
    /// with NALUs in the H264 packet buffer supplied
    /// for caption message extraction.
    void setNALLengthFieldSize(uint32_t inSize);

    /// This method extracts caption message from the
    /// H.264 packet buffer for video data. The buffer
    /// is assumed to contain length prefixed NALUs
    /// corresponding to same decoding time.
    /// Returns true if a message is extracted
    bool extractCaptionMsgFromH264Pkt(
        adbe::hds::MediaMessage& outCaptionMsg,
        const uint8_t* pH264Buffer,
        const uint32_t h264PktSize,
        uint64_t dts,
        uint64_t ctts);
};
```

MPEGParser - Support for Ad signalling (New in 3.0)

The MPEGParser utility has been improved to parse the SCTE-35 markers present in the TS stream. The utility parses the SCTE-35 markers and reads the required Ad cue information from them. This Ad cue information is used to create a `SpliceCueInfo` object, which is then pushed into `Packager` using the new API `IPackager::insertSpliceCue`. The `Packager` then inserts the Ad cue information into the F4M file. Ad signaling is only supported for live packaging.

Note the following rules while parsing the SCTE-35 markers:

1. Only `splice_insert()` messages with an `out_of_network_indicator` flag set to 1 (splice out messages) are supported. Messages having a value of 0 for this flag (splice in messages) are ignored.
2. Only `splice_insert()` messages that provide a defined `break_duration()` are supported. If `break_duration()` is not defined, the message will be ignored. Without an up-front break duration value there is no good way to resolve Ad placements to fill the break.
3. The `auto_return` flag in a `break_duration()` is ignored. An auto-return from insertion stream to the network feed should happen at the end of the break duration.
4. `splice_insert` messages having the `splice_immediate_flag` as 1 and no defined `splice_time()` are ignored. Currently `splice_insert()` messages with explicit `splice_time()` are supported.
5. The `splice_insert()` messages with the `cancelled` parameter set to true are ignored.

API changes

A new method, `insertSpliceCue()` has been added to the `IPackager` class.

```
class IPackager
{
    ...
    ErrorCode insertSpliceCue(const SpliceCueInfo& spliceCueInfo);
    ...
};
```

This new method takes in the Ad cue information in the form of `SpliceCueInfo` and stores it in a list. When the `Packager` module updates the F4M file, it adds a `<Cue>` tag corresponding to each of the `SpliceCueInfo` in the F4M file.

Note: The F4M specification has also been updated to include a new `@version` attribute for the `<manifest>` element. The absence of the `@version` attribute is equivalent to setting the `@version` to 1.0 for backward compatibility. For Ad Signaling to work, the Adobe Primetime player requires `@version` to be set to 3.0. Also, the Adobe Primetime player requires the Bootstrap information to be kept inside the F4M for ad signaling to work.

The following example shows an F4M file having the `<Cue>` tags (one per `SpliceCueInfo`):


```
<manifest xmlns="http://ns.adobe.com/f4m/1.0" version="3.0">
  <bootstrapInfo id="bootstrap1234">Bootstrap Information</bootstrapInfo>
  <cueInfo id="cueInfo1234">
    <cue type="SpliceOut" id="1" time="266.061" duration="30.5"
      programId="2354" availNum="1" availsExpected="5" />
  </cueInfo>
  <media streamId="stream1_500kbps" url="stream1_500kbps"
    bootstrapInfoId="bootstrap1234" cueInfoId="cueInfo1234" />
</manifest>
```

SpliceCueInfo Structure

The following code snippet describes the SpliceCueInfo structure and its members:

```
struct SpliceCueInfo
{
    fms::String cue;    //Required for SCTE35Mode packet, contains base64
                        encoded binary SCTE packet

    CueType cueType;    //Used to indicate blackout or ad cue.

    bool immediate;    //"Splice Immediate Mode" used for the early
                        termination of breaks

    fms::String type;    //The Ad signalling event type which
                        //must be "SpliceOut".

    fms::String id;    //A unique identifier for this Ad
                        //Signalling event.

    double time;        //The stream's presentation time in
                        //fractional seconds at which point the
                        //splice out (Ad insertion) should
                        //occur.

    double duration;    //The splice duration in fractional
                        //seconds.

    int    programId;    //Optional; Identifier for the program
                        //content this splice applies to. When
                        //set to 0, it is ignored.

    int    availNum;    //Optional; Index of this specific
                        //avail within the total set of avails
                        //within the stream being packaged
                        //identified by the programId. When set
                        //to 0, it is ignored.
```

```

int    availsExpected;
        //Optional; the expected total number
        //of avails within the stream being
        //packaged (identified by programId).
        //When set to 0, it is ignored.
};

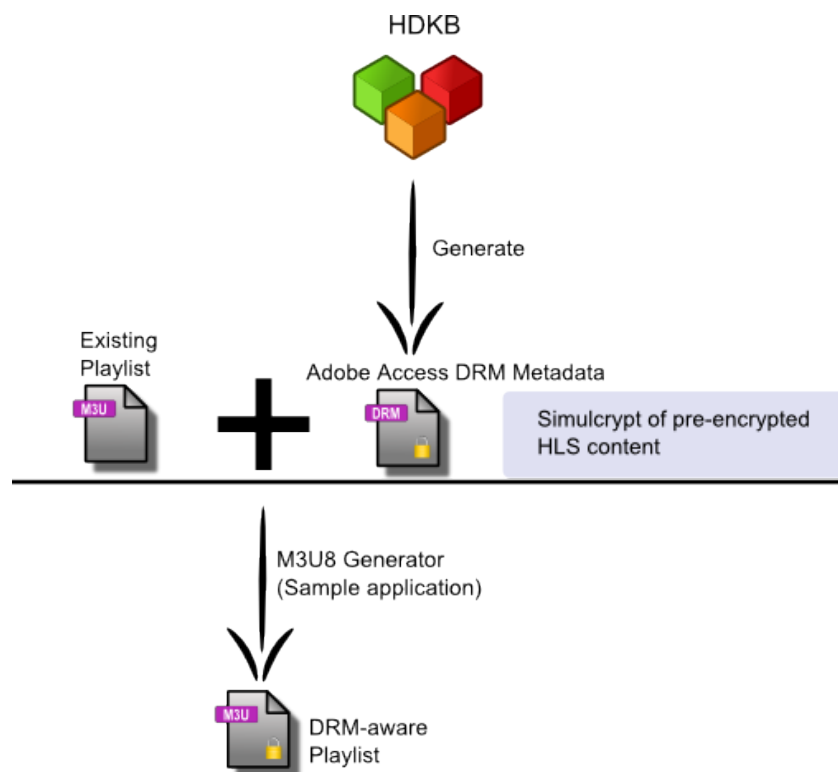
```

You need to populate the `SpliceCueInfo` structure with relevant values so that it can be passed to the Packager module.

M3U8Generator

The M3U8Generator is a command line tool that takes an existing playlist file (M3U8) and encryption keys for pre-encrypted HLS content and produces a new M3U8 file, augmented with the information required for Adobe Access.

Figure3: Use M3U8Generator to produce DRM-aware playlists



The command line interface for `m3u8generator` is similar to `offlinepackager`. Options can be set either on the command line or through an XML configuration file. Options set at the command line override any options you set in the configuration file.

The usage syntax is as follows:

```
m3u8generator --input-file=<file> --[option]=[arg]...
```

You can provide one or more of the following options:

Option	Description
--conf-file <arg>	The configuration file that contains settings for the packaging process.
--input-file <arg>	The path to the input M3U8 file. The path can be absolute or relative.
--input-key-dir <arg>	The path to a directory containing the encryption key(s) listed in the M3U8 file. The path can be absolute or relative.
--output-file <arg>	The path to the destination M3U8 file. The path can be absolute or relative.
--external-metadata <arg>	Specifies whether to generate an external metadata file or include the metadata in the M3U8. If true, the metadata is written to a file output-file.m3u8.
--key-server-url <arg>	URL of the Adobe Access Key Server that handles remote key delivery to iOS devices. This option is required if the policy requires remote key delivery.
--key-server-cert <arg>	The path to DER-encoded key server certificate file for remote key delivery to iOS devices. The path can be absolute or relative.
--license-server-url <arg>	The URL of the license server that handles license acquisition for this content.
--transport-cert <arg>	The DER encoded transport certificate file. The file argument can be absolute or relative like the input-file argument.
--license-server-cert <arg>	The DER encoded license server certificate file used for content protection. The path can be absolute or relative.
--packager-credential <arg>	The PFX file containing the packager's protection credentials. The path can be absolute or relative.
--credential-pwd <arg>	The password string used to secure the packager credentials.
--content-id <arg>	The content ID used with the common key to generate the Content Encryption Key (CEK).
--policy-file <arg>	The file containing the policy for this content. The path can be absolute or relative.
--license-server-credential <arg>	The PFX file containing the license server's protection credentials. The path can be absolute or relative.
--license-server-credential-pwd <arg>	The password string used to secure the license server credentials.
--recipient-cert <arg>	The DER encoded recipient certificate file. The argument can be absolute or relative like the input-

	file argument.
--enable-phds <arg>	Specifies whether to enable the PHDS feature.
--generate-chained-license <arg>	Specifies whether to generate chained license or not.
--spliceinfo-file <arg>	New in 3.0. Specifies the path to the XML file containing the information regarding the splice information and the sequence number of the TS after which it should be included. See M3U8Generator - Support for Ad signalling (New in 3.0) .
--enable-license-rotation	New in 3.2 No value parameter to enable the license rotation
--license-info-file	New in 3.2 This file will contain the additional information required for license generation. User can specify additional parameter for the FAXS meta-data information that is to be generated for individual license.
--metadata-relative-position	New in 3.2 Default value is 1. Specifies the relative backward position (in terms of number of segment) of #EXT-X-FAXS-CM tag with respect to the associated #EXT-X-KEY in the destination m3u8 file.

Note: In case of relative path, when passed on the command line, the path is relative to the current working directory. If passed through a configuration file, the path is relative to the directory containing the configuration file.

M3U8Generator - Support for HLS Ad signalling (New in 3.0)

M3U8Generator now accepts a new parameter, `--spliceinfo-file`, which can point to the XML file containing the information regarding the splice information and the sequence number of the TS after which it should be included.

The following example shows a sample splice information file:

```
<seqspliceinfos>
  <seqspliceinfo>
    <adtype>midroll</adtype>
    // The adType of the splice. The valid values are preroll,
    // midroll and postroll. If absent, the adtype
```

```

        // defaults to midroll.

<seq>4</seq>
// The TS sequence number after which the Splice Cue
// should be inserted in M3U8. Used only when value
// of adType is midroll or <adtype> is absent.

<type>SpliceOut</type>
// The Ad signalling event type which must be "SpliceOut".

<id>1</id>
// A unique identifier for this Ad Signalling event.

<time>20.22</time>
// The stream's presentation time in fractional
// seconds at which point the splice out (Ad
// insertion) should occur.

<duration>10.45</duration>
// The splice duration in fractional seconds.

<programId>1</programId>
// Optional; Identifier for the program
// content this splice applies to.

<availNum>2</availNum>
// Optional; Index of this specific avail within
// the total set of avails within the stream
// being packaged (identified by programId).

<availsExpected>10</availsExpected>
// Optional; the expected total number of avails within
// the stream being packaged (identified by programId).
    </seqspliceinfo>
</seqspliceinfos>

```

The previously mentioned example will add the following CUE tag after TS with the sequence number 4:

```
#EXT-X-CUE:TYPE=SpliceOut,ID=1,TIME=20.22,DURATION=10.45,PROGRAM-ID=1,
AVAIL-NUM=2,AVAILS-EXPECTED=10
```

If the sequence number provided is less than the first sequence number of the M3U8, the splice information will act as a pre-roll splicing information. If the sequence number provided is more than the last sequence number of the M3U8, the splice information will act as post-roll splicing information.

M3U8Generator changes for HLS License Rotation support (New in 3.2)

To support HLS license rotation, the following application parameters are added in in HDKB sample utility application `m3u8generator` to support license rotation:

1. `enable-license-rotation` (optional): Specifies whether to enable license rotation or not.
2. `license-info-file` (optional): This file contains the additional information required for license generation. AAXS meta data i.e. the key URL present in EXT-X-KEY tag is used as the primary id to associate information present in the file with the individual key tags and hence generate license associated with key tag. You need not specify this additional information for key tag. If you do not specify this additional information for any key tag then, the default information is used. The following additional information can be specified in the license information file:
 - a. `license-content-id` (optional): Content id must be changed per license because it is used to generate license id. If it is not specified, a content id is generated for associated license by adding a sequential suffix to the value of content id specified using the tool command line parameter `content-id`.
 - b. `license-policy-file` (optional): You can specify a list of policies for every license using the `license-policy-file` parameter. If you do not specify the policy file for any license, the default policy file specified using the tool command line parameter `policy-file` is used.
3. `metadata-relative-position` (optional): Default value is 1. Specifies the relative backward position (in terms of number of segment) of #EXT-X-FAXS-CM tag with respect to the associated #EXT-X-KEY in the destination m3u8 file. It will help the client application to fetch the license in advance before it reaches to the actual key URI tag.

Here are the sample contents of a sample license-info-file:

```
<license-info-list>
  <license-info key-file="../../xxxxx/input/dme/key_lr_2.bin">
    <license-content-id>ContentId2</license-content-id>
    <license-policy-file>../resources/input/dme/policy_Auth.pol</license-
policy-file>
  </license-info>
  <license-info key-file="../../xxxxxx/input/dme/key_lr_3.bin">
    <license-content-id>ContentId3</license-content-id>
    <license-policy-file>../xxxxx/input/dme/policy_custom1.pol</license-
policy-file>
  </license-info>
</license-info-list>
```

The following is an example of an m3u8 input file:

```
#EXTM3U
#EXT-X-TARGETDURATION:6
#EXT-X-VERSION:2
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:6,
#EXT-X-KEY:METHOD=AES-
128,URI="crypt0.key",IV=0xb227027e822195fcb429ccb6177892f2
fileSequence0.ts
#EXTINF:6,
#EXT-X-KEY:METHOD=AES-
128,URI="crypt1.key",IV=0xd96e330f93dd74d0f6f5b4dcc30e0270
fileSequence1.ts
#EXTINF:6,
#EXT-X-KEY:METHOD=AES-
128,URI="crypt2.key",IV=0x9afb569a96ea091a66b7c37e85b494b
fileSequence2.ts
#EXTINF:6,
#EXT-X-KEY:METHOD=AES-
128,URI="crypt3.key",IV=0x170f5b6686e377cc0d871b9918a140d4
fileSequence3.ts
#EXT-X-ENDLIST
```

The following is an example of the corresponding m3u8 output file with license rotation output:

```
#EXTM3U
#EXT-X-TARGETDURATION:6
#EXT-X-VERSION:2
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:6,
#EXT-X-FAXS-CM:MIIYJwYJKoZIhvcNAQcCoIIYGDCGGBQC...WZoZ8w0I6iDCA==
#EXT-X-KEY:METHOD=AES-
128,URI="faxes://faxes.adobe.com",IV=0xcd8de0a2fd17eeb478fcde5fb761a274,CMSHa1Hash=0xa54371d6b78101b810a2bad8ba1f00d5a91edc59
fileSequence0.ts
#EXTINF:6,
#EXT-X-FAXS-CM:MIIYJwYJKoZIhvcNAQcCoIIYGDCGGBQC...swwtG0Pl0sMJmQ==
#EXT-X-KEY:METHOD=AES-
128,URI="faxes://faxes.adobe.com",IV=0x66756872bf40a5300b909f6fbaec94f8,CMSHa1Hash=0x4ebb4df9530fe37a5b5c10171c8bc123e6eae796
fileSequence1.ts
#EXTINF:6,
#EXT-X-FAXS-CM:MIIYJwYJKoZIhvcNAQcCoIIYGDCGGBQC...u6Gdbq/UIqyAAw==
#EXT-X-KEY:METHOD=AES-
128,URI="faxes://faxes.adobe.com",IV=0x7555e546827004da243d935ce9f37415,CMSHa1Hash=0x5a39688f98ac16678b53489e0730ff9dcd8fc554
fileSequence2.ts
#EXTINF:6,
```

```
#EXT-X-FAXS-CM:MIIYJwYJKoZIhvcNAQcCoIIYGDCCGBQC...Gk2j95x/fyejvg==
#EXT-X-KEY:METHOD=AES-
128,URI="faxes://faxes.adobe.com",IV=0xf3337eca99b5786ce8461d983f478934,CMSHa1Hash=0xd2c973f7295add36941ddc041fd7ea3431ebfc7b
fileSequence3.ts
#EXT-X-ENDLIST
```

M3U8Generator usage workflow

The following sections provide steps in using the M3U8Generator sample application:

Step 1: Pre-encrypting content

Third party utilities, such as Apple's [mediafilesegmenter](#), can be used to generate encrypted HLS content. For example, to encrypt content with a single key:

```
mediafilesegmenter -k <key-directory> <filename>
```

To encrypt content with key rotation

```
mediafilesegmenter -k <key-directory> -key-rotation-period
<rotation-frequency> <filename>
```

The M3U8 file and key directory generated during content encryption need to be provided to the M3U8 Generator sample application.

Step 2: Adding Adobe Access Metadata

To run the M3U8 Generator, use the following command:

```
m3u8generator --conf-file config.xml
```

The following examples show the parameters needed in config.xml for various use cases.

Use case: Licenses obtained from Adobe Access license server, with local or remote key delivery

```
<offline>
  <input-file>in.m3u8</input-file>
  <input-key-dir>key-directory</input-key-dir>
  <output-file>out.m3u8</output-file>
  <content-id>unique-content-id</content-id>
  <license-server-url>http://host:port</license-server-url>
  <license-server-cert>license-server.der</license-server-cert>
  <transport-cert>transport.der</transport-cert>
  <packager-credential>packager.pfx</packager-credential>
  <credential-pwd>packager-password</credential-pwd>
  <policy-file>policy.pol</policy-file>
</offline>
```

Note: key-server-url parameter is also required for remote key delivery.

Use case: PHLs (local key delivery only)

```

<offline>
  <input-file>in.m3u8</input-file>
  <input-key-dir>key-directory</input-key-dir>
  <output-file>out.m3u8</output-file>
  <content-id>unique-content-id</content-id>
  <license-server-url>http://host:port</license-server-url>
  <license-server-cert>license-server.der</license-server-cert>
  <transport-cert>transport.der</transport-cert>
  <packager-credential>packager.pfx</packager-credential>
  <credential-pwd>packager-password</credential-pwd>
  <policy-file>phds/static/phds_24hr_policy.pol</policy-file>
  <enable-phds>true</enable-phds>
  <license-server-credential>
    license-server.pfx
  </license-server-credential>
  <license-server-credential-pwd>password</license-server-credential-pwd>
  <recipient-cert>phds/sd/ADBEXP.cer</recipient-cert>
</offline>

```

Use case: Embed chained leaf license, with local or remote key delivery

```

<offline>
  <input-file>in.m3u8</input-file>
  <input-key-dir>key-directory</input-key-dir>
  <output-file>out.m3u8</output-file>
  <content-id>unique-content-id</content-id>
  <license-server-url>http://host:port</license-server-url>
  <license-server-cert>license-server.der</license-server-cert>
  <transport-cert>transport.der</transport-cert>
  <packager-credential>packager.pfx</packager-credential>
  <credential-pwd>packager-password</credential-pwd>
  <policy-file>chaining.pol</policy-file>
  <enable-phds>true</enable-phds>
  <license-server-credential>
    license-server.pfx
  </license-server-credential>
  <license-server-credential-pwd>password</license-server-credential-pwd>
  <generate-chained-license>true</generate-chained-license>
</offline>

```

The key-directory element refers to the directory containing the encrypted keys. This is the same directory that you have used along with the -k option while using mediafilesegmentor in Step 1.

Note: key-server-url parameter is also required for remote key delivery.

Use case: Embed machine-bound license with local or remote key delivery

```

<offline>
  <input-file>in.m3u8</input-file>

```

```
<input-key-dir>key-directory</input-key-dir>
<output-file>out.m3u8</output-file>
<content-id>unique-content-id</content-id>
<license-server-url>http://host:port</license-server-url>
<license-server-cert>license-server.der</license-server-cert>
<transport-cert>transport.der</transport-cert>
<packager-credential>packager.pfx</packager-credential>
<credential-pwd>packager-password</credential-pwd>
<policy-file>remote.pol</policy-file>
<enable-phds>true</enable-phds>
<license-server-credential>
    license-server.pfx
</license-server-credential>
<license-server-credential-pwd>password</license-server-credential-pwd>
<recipient-cert>machine-cert.der</recipient-cert>
<key-server-cert>key-server.der</key-server-cert>
<key-server-url>https://host:port/faxsks/tenant/key</key-server-url>
</offline>
```

Note: key-server-cert and key-server-url parameters are required parameters for remote key delivery.

Sample code

The sample code in the `m3u8manipulator` and `m3u8generator` projects illustrates how to modify an M3U8 file for pre-encrypted content to enable the Adobe Access support. The `M3U8Generator` project contains an implementation of the M3U8 Generator command line interface and also illustrates how to construct the DRM metadata using the HDKB APIs. The `m3u8manipulator` project illustrates how to parse an M3U8 file, add Adobe Access information, and write the modified M3U8 file.

Also, to support ad insertion, the M3U8 files will be modified to insert cue points.

For more information on generating an Adobe Access-enabled M3U8 file, see the *HDKB 3.0 Overview* document.

Copyright

© 2013 Adobe Systems Incorporated. All rights reserved.

Adobe HTTP Dynamic Streaming SDK Broadcast Sample Applications
Edition 3.0

This guide is licensed for use under the Creative Commons Attribution Non-Commercial 3.0 License. This License allows users to copy, distribute, and transmit the guide for noncommercial purposes only so long as (1) proper attribution to Adobe is given as the owner of the guide; and (2) any reuse or distribution of the guide contains a notice that use of the guide is governed by these terms. The best way to provide notice is to include the following link. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/us/>.

Adobe and the Adobe logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Windows is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. Red Hat is a trademark or registered trademark of Red Hat, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

RSA Security, Inc.

This product contains either BSAFE and/or TIPEM software by RSA Security, Inc.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.