



Adobe HTTP Dynamic Streaming SDK Broadcast API Overview

3.5

Contents

Overview	4
Support for HDS	4
Support for HLS	5
What is new in HDKB 3.5	5
External CEK support.....	5
What is new in HDKB 3.2	5
Support for license rotation.....	5
Support for regional blackouts	6
What is new in HDKB 3.1	6
Introduction of HDS Validator Tool.....	6
What is new in HDKB 3.0	6
Support for Ad Signalling	6
In HDS.....	6
In HLS	8
Support for Closed Captioning (CC)	8
Support for Trick Play.....	10
API details	11
IFragmenter	11
Key Rotation.....	12
License Rotation.....	12
Support for regional blackouts	13
Support for external CEK.....	13
PHDS (Protected HTTP Dynamic Streaming).....	14
Chained License Generation	15
Fragmentation.....	16
IEncryptor.....	19
IFragmentExtractor	21
IMediaSource, IFileSource, ISampleIterator, and IMsgInfoIterator.....	21
IFragment.....	21

MediaMessage	22
IMsgInfolterator	22
IMoovBuffer	23
ILogger and IMultiLogger	23
KeyGenerator	24
Usage scenarios for HLS-based content protection.....	24
Making HLS content DRM-enabled.....	24
Generating DRM metadata	25
Full DRM protection.....	26
HDKB with PHLS	27
Generating an Adobe Access-enabled M3U8 file	28
Key URLs.....	29
Local Key Delivery	29
Remote Key Delivery.....	29
Key Rotation.....	29

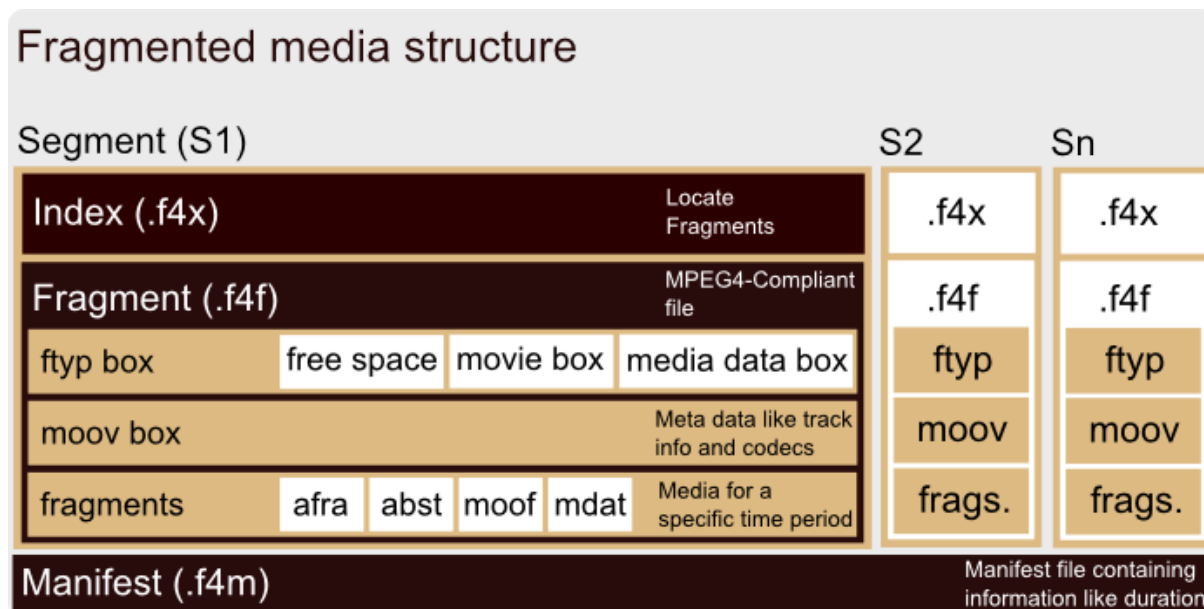
Overview

Adobe HTTP Dynamic Streaming SDK Broadcast (HDKB) APIs enable developers to create/enhance video streaming solutions supporting the most common HTTP streaming formats including HTTP Dynamic Streaming (HDS) and HTTP Live Streaming (HLS). In the case of HDS, content can be packaged, encrypted, and delivered to supported clients. In case of HLS, the content can be enhanced by making them DRM and ad-signaling enabled. This document provides an overview of how HDKB can be used to create/enhance HDS and HLS content.

Support for HDS

HDKB supports creating fragmented media for HDS streaming. Fragmented media consists of one or more segments and a manifest (.f4m) file. Each segment consists of one fragment file (.f4f) and one index (.f4x) file. Each fragment file is an MPEG4-compliant file with an ftyp box, a moov box and one or more fragments. ftyp box and moov box are container boxes in the hierarchical structure of MPEG4 files. The ftyp box contains file type and compatibility information. A moov box contains metadata information related to the file, for example, the track information (including the codecs used) and the total duration.

Figure 1: Fragmented media structure



Each fragment represents the media for a specific time period and has samples from all the tracks in the media for that time period. The samples related to the tracks for that duration must be in the same fragment. Each fragment has four MPEG4 boxes – afra, abst, moof, and mdat. The index file is used to locate a particular fragment in the segment.

Note: See the *HDKB API documentation* available in the doxygen directory included in the SDK for more details.

Support for HLS

HDKB along with Adobe Access can be used to set up a complete DRM-enabled content delivery workflow for delivering encrypted on-demand and live content to devices supporting HLS. HDKB provides tools to convert pre-encrypted HLS content into Adobe Access supported DRM-enabled HLS content. This new DRM setup enables the delivery of DRM key locally or remotely to iOS clients based upon the specified DRM policies.

HDKB also supports [Ad signalling for HLS](#).

For more information of Adobe Access, see the [Adobe Access documentation](#).

What is new in HDKB 3.5

HDKB 3.5 supports the following feature:

External CEK support

This feature allows the HDS content packager and License Server to use a shared secret to derive or fetch a CEK_ID and CEK pair for encryption. The shared secret may be an algorithm, look-up table, or any other entity.

In this new mechanism, the content packager uses the shared secret to derive/fetch a CEK_ID and CEK pair. The content is encrypted, as usual, using the CEK. The associated CEK_ID is embedded inside the DRM metadata, instead of the CEK.

On the other side, the license server uses the CEK_ID to derive/fetch the CEK by applying the shared secret. The derived/fetched CEK is included in the license that is returned to the HDS client.

The license that the HDS client receives from the license server is identical to what it normally receives using the earlier mechanism (in which CEK is embedded inside DRM metadata in encrypted form).

For details around HDKb API changes for the feature, see Support for external CEK.

What is new in HDKB 3.2

HDKB 3.2 supports the following features:

Support for 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 and uses them to decrypt and play the packaged HDS content.

For more information, see License Rotation.

Support for regional blackouts

Regional blackout in the broadcast industry refers to the non-availability of content in a specific geographical region of the media market.

For more details, see [Support for regional blackouts](#).

What is new in HDKB 3.1

Introduction of HDS Validator Tool

Content validation for HTTP dynamic streaming is a process that verifies whether the HDS content created using the third-party tools is in compliance with Adobe's standard. HDS Validator is a command-line utility that is used for HDS content validation. For more information, see the [Content Validation Tool Getting Started](#) guide.

What is new in HDKB 3.0

The following features are now supported in HDKB 3.0:

1. [Ad signalling](#)
2. [Closed captioning](#)
3. [Trick play](#)

Support for Ad Signalling

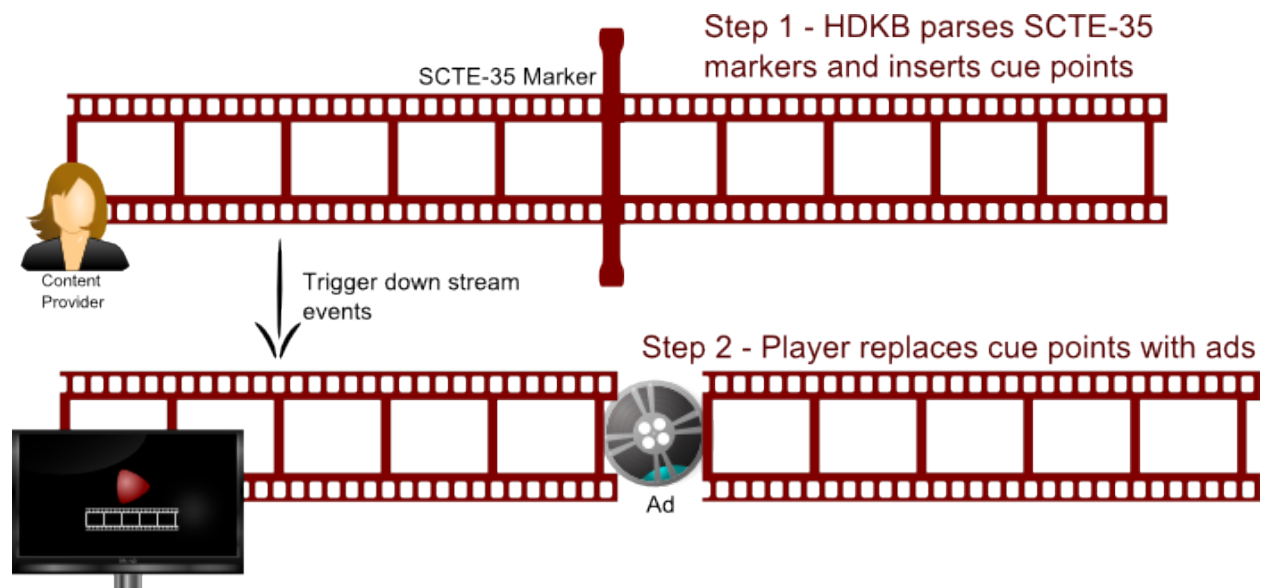
HDKB enables content providers and broadcasters to monetize live streaming with accurate and personalized ad insertion. HDKB allows for the addition of cue points or ad markers in the F4M file (for HDS) and M3U8 file (for HLS), which is then used by the player to dynamically insert ad content.

Ad signalling is supported for both HDS and HLS source streams.

In HDS

The new `IPackager` API allows you to provide cue point information to the `Packager`, which in turn embeds that information in the F4M file. However, it doesn't provide any implementation to extract any information from the source stream. The `MPEGParser` sample utility has been updated to parse the SCTE-35 markers present in the incoming TS content to identify the cue points in the stream, and pass that information to the `Packager` module via the new API.

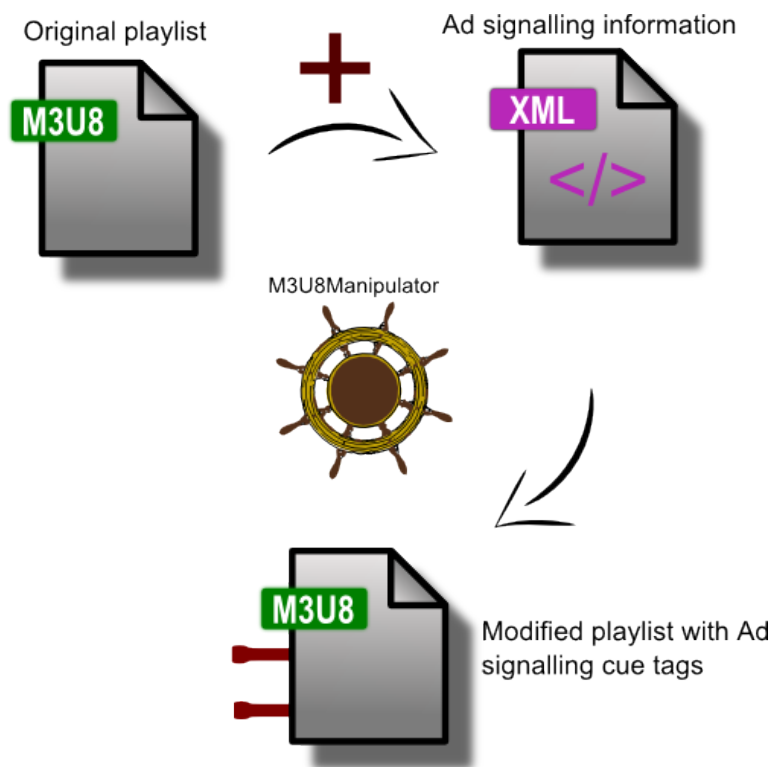
Figure 2: Ad signalling in HDS



In HLS

In the case of HLS, *M3U8Manipulator* can be used to add the required cue points in the M3U8 file. *M3U8Manipulator* is capable of modifying the playlist by inserting ad signaling cue tags.

Figure 2a: Ad signalling in HLS



Note: See the *HDKB 3.1 Sample Applications* document for more information.

Important: The player supporting HDKB's ad signaling is an enhanced version of the Adobe Primetime player. Please contact Adobe if you want to be part of the early adopters group to test the Ad signaling workflow.

Note: For changes made in the HDKB sample applications to support Ad insertion, see the *HDKB 3.0 Sample Applications* document.

Support for Closed Captioning (CC)

Closed captioning allows the content provider to display text overlaid on the video content to provide information necessary to allow users who are deaf or hard of hearing to understand the content of the video. Closed captioning is primarily used as an accessibility feature to represent the audio portion of the video content (and any additional information necessary to understand the content such as speaker identification) as it occurs. The term "closed" in closed captioning indicates that the captions can be

turned on or off by the viewer, which requires that the video player contain the capability to decode and render the captioning data.

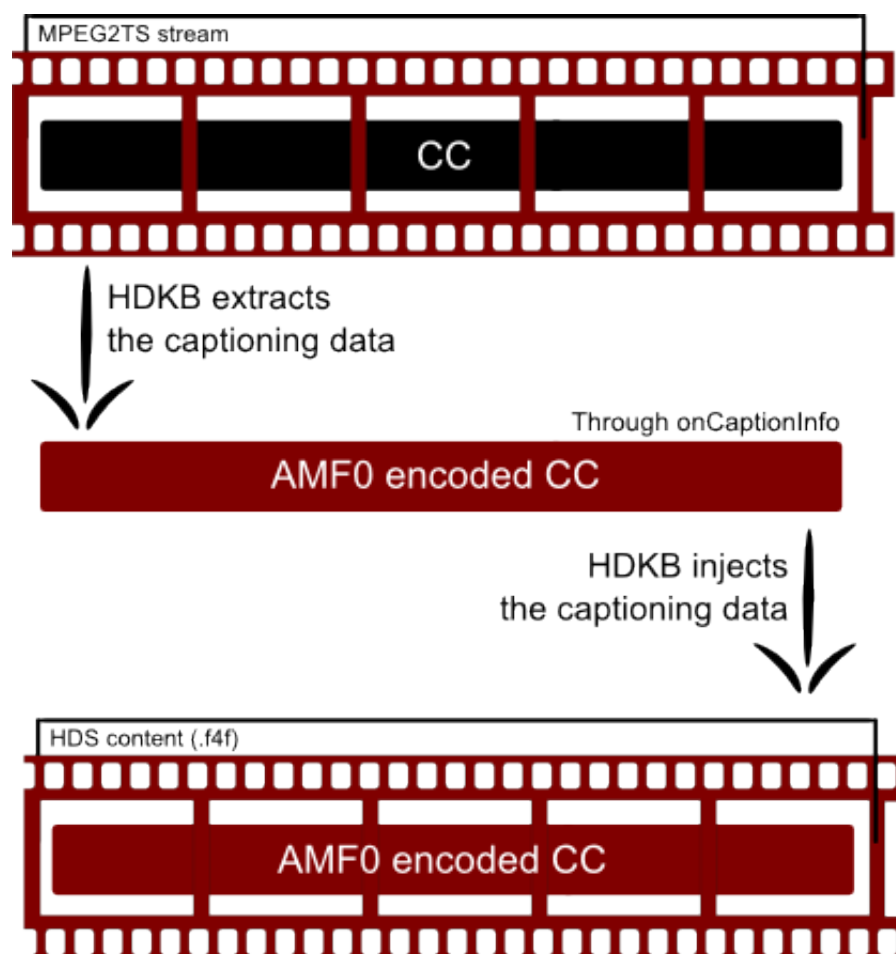
HDKB supports industry standard [608/708 closed captioning](#) for video streaming to the following output channels/devices:

- [Project Primetime](#)

HDKB can be part of a complete solution for packaging and embedding 608/708 closed captioning data in HDS stream. The captioning data embedded in the HDS stream can be playable by HDS clients supporting the Project Primetime SDK.

HDKB extracts the 608/708 captioning data from the MPEG2TS source stream, wraps the captioning data in a special AMF0 encoded data message called “onCaptionInfo” message, and embeds the message into the HDS stream. See the *onCaptionInfo specification document* included in the HDKB bundle for more information.

Figure 4: How HDKB handles captioning data



Note: See the *HDKB 3.1 Sample Applications* document for more information.

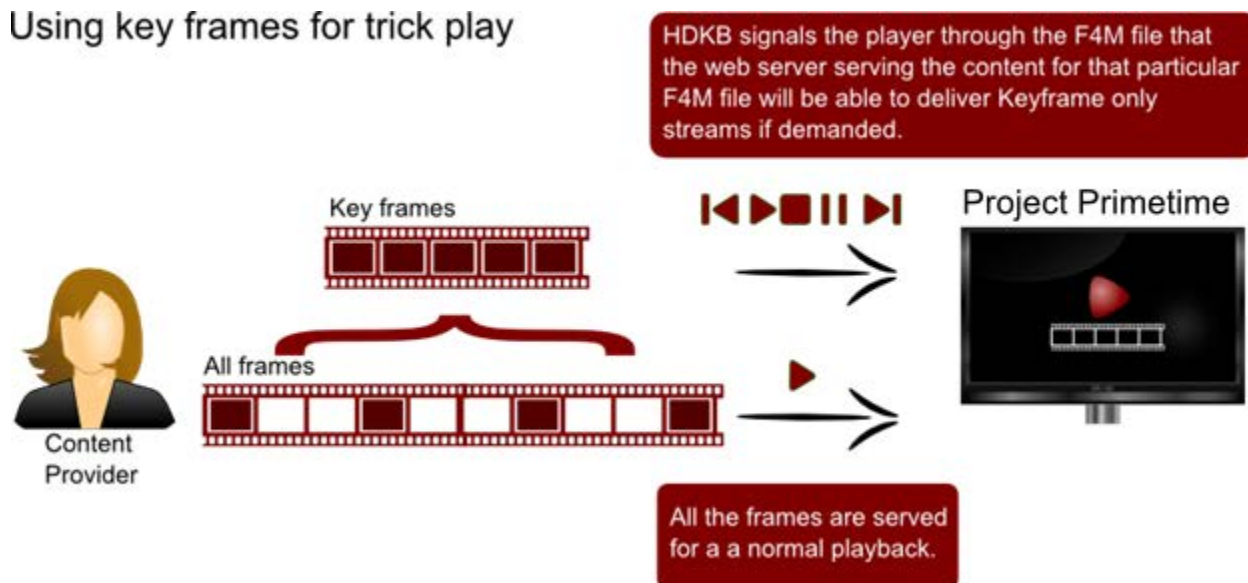
Support for Trick Play

Video on demand systems (supporting downloading and streaming) enable consumers to use pause, fast forward, rewind and other similar functionalities. These functions are called trick modes and execution of these functions is called trick play.

There are certain trick modes that the Adobe Primetime player supports without any additional support from HDKB. For instance, a slow forward trick play can be performed by the player without any special support from the server. However, in order to achieve fast forward and fast rewind trick modes, the player will require additional support from the web server to serve key-frame only fragments. The key frame only fragment contains the Video AVC (Advanced Video Config) and the first video keyframe, both of which are read from the corresponding actual fragment.

Figure 5: Using key frames for trick play

Using key frames for trick play



The `OfflinePackager` sample application has been updated to generate an F4M file that instructs the Adobe Primetime player that the content represented by the F4M file can be served as keyframe only variant, which can be used by the player to perform trick play.

Note: See the *HDKB 3.1 Sample Applications* document for more information.

API details

The Adobe HTTP Dynamic Streaming SDK Broadcast is composed of the following basic interfaces:

- `IFragmenter`
- `IEncryptor`
- `IFragment`
- `IFragmentExtractor`
- `ILogger`
- `IMultiLogger`
- `IMoovBuffer`

In addition to these interfaces, the SDK contains the following utility interfaces to standardize the way the media files are read and iterated:

- `IFileSource`
- `IMediaSource`
- `ISampleIterator`
- `IMsgInfoIterator`

The SDK uses two basic classes to hold actual message data and information related to the message:

- `MediaMessage`
- `MessageInfo`

The various objects used in the SDK (for example, `IFragmenter` or `IFragment`) are used through smart pointers of type `boost::intrusive_ptr`. The names of smart pointers of various interfaces are formed by adding the suffix `Ref` to the interface name (for example, `IFragmenterRef` and `IEncryptorRef`). These objects are better passed by value and are automatically destroyed when they go out of scope. The `MediaMessage` object also holds a smart pointer to the actual data buffer apart from its metadata. The `MediaMessage` object is passed by value and is copied. The buffer containing the data gets deleted when all the `MediaMessage` objects referring to it go out of scope. All these objects (the `Ref` objects as well as the `MediaMessage` objects) are assignable and copy-constructible and hence can be used with STL containers.

IFragmenter

`IFragmenter` is the main interface for creating fragments from a stream of messages. The messages can originate from a live stream or from a file. This interface supports creation of fragments with or without content protection.

First, the application calls the function `createFragmenter` to create a `Fragmenter` object, which implements the `IFragmenter` interface. Some flags can be passed to the `Fragmenter` to set its mode. An object ID can also be passed to `createFragmenter`. The object ID is a string that helps the user to

identify the object. This object ID is sent to the logger when the SDK calls `logMessage`. If there are multiple streams being packaged simultaneously, the object ID field can be used to distinguish between log messages coming from various streams. This can be done by setting the object ID of each object to the name of the stream the object is associated with. For more details, refer to the HDKB sample application code or the description of the `ILogger` class.

If the application is a live-streaming application, it has to set the `kFMLive` flag. If it is a restart scenario, the flag `kFMRestart` also needs to be set. Then the application (optionally) sets the fragmenter configuration using the function `setConfiguration`. To enable content protection, the user will have to supply the parameters required for encryption in the `FragmenterConfig` structure that is passed to the `setConfiguration` function. For details about the `FragmenterConfig` and `EncryptionConfig` structures please refer to the doxygen documentation included within this SDK.

The `EncryptionConfig` structure provides encryption-related parameters (such as the Content Encryption Key, License Server URL, and Transport Certificate) to `IFragmenter`. The `EncryptionConfig` also contains parameters to enable and use Key Rotation, Protected HTTP Dynamic Streaming (PHDS), and the Chained License generation features of the SDK.

Key Rotation

During a typical packaging with encryption enabled, the media content is encrypted using the Content Encryption Key (CEK). In addition, the media player obtains a license containing the CEK to consume the content. When Key Rotation is enabled, the key used to encrypt the samples can be changed so that each rotation key is only used to encrypt a portion of the content. However, the media player still only needs to obtain a single license to consume the content. For more information, see the [Adobe Access 4.0 Protecting Content Guide](#).

License Rotation

The `IFragmenter` class has introduced a new function, `updateLicense()`. This function updates the license information in the `IFragmenter`. You can call this function at any point of time with different license information. The license rotation can take place within the fragment and at the fragment boundaries.

This API need not be called for the very first license, because the initial license information can be provided through initial configuration parameters. The initial configuration parameters are provided to `IFragmenter` through `FragmenterConfig::m_EncryptionConfig` structure using `IFragmenter::setConfiguration()` API.

During the process of pushing messages to the HDKB, whenever you need to rotate the license information, call `IFragmenter::updateLicense()` API. A new structure `LicenseInfo` is introduced to pass the license information to the `updateLicense()` API. **For more information, refer to the Doxygen documentation provided with the SDK.**

The following is the updated `IFragmenter` class.

```
namespace adbe
```

```

{
    namespace hds
    {
        class IFragmenter
        {
            ...
            /// This function updates the license to be used for encrypting
            next media messages.
            /// \param [in] licenseInfo license information for the license.
            /// \return Success or error code indicating the error
            ErrorCode updateLicense(const LicenseInfo& licenseInfo);
            ...
        };
    }
}

```

Support for regional blackouts

To enable regional blackouts, HDKB samples parse the input SCTE-35 start and end blackout signals and insert cue tags in the manifest file. It also rotates the licenses at the blackout boundaries to restrict clients from fetching the content in blackout regions.

The existing API, `adbe::hds::ErrorCode IPackager::insertSpliceCue(const SpliceCueInfo & spliceCueInfo)` has been updated.

This API takes the `SpliceCueInfo` structure filled using the information extracted from the SCTE-35 message for the contents being packaged and inserts the corresponding <cue> elements in the manifest file.

The Struct, `SpliceCueInfo`, gets the following additional members:

```

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

```

Support for external CEK

To support the External CEK feature, the following changes are made to its HDKb interfaces:

- The HDKb encryption configuration (`adbe::hds::EncryptionConfig`) to generate DRM metadata (using `IEncryptor::getDRMMetadata()`, `IEncryptor::getDRMAdditionalHeader()` APIs) now includes the optional parameter `fms::String cekId`. This parameter serves as the identifier for CEK. The presence of the parameter value indicates that the content is to be encrypted using the External CEK feature.

Note: While specifying the `cekId` parameter, the common key can't be specified. In other words, `bool isCommonKey` should be set to `false`.

- An additional parameter `fms::String cekId` is available in the structure `adbe::hds::LicenseInfo` to support external CEK feature with license rotation.

PHDS (Protected HTTP Dynamic Streaming)

For packaging with encryption enabled, it is possible to implement a workflow where a media player need not contact a license server to obtain a license, since the license is embedded into the DRM metadata accompanied with the content. To support the workflow, the application must provide PHDS parameters (for example, the License Server Credential and Recipient Certificates) using the `EncryptionPHDSConfig` structure, which is a child structure of `EncryptionConfig`. These parameters are used to generate a license out-of-band and embed the license into the `DRMAdditionalHeader` element of the Media Manifest file. This information can be used by a media player without contacting a license server. As part of the PHDS configuration, you can also provide the SWF digests of Flash and AIR applications which may be used to play the content. These are embedded in the license, which enables Flash Player to verify that the current SWF is approved by the content provider.

EncryptionPHDSConfig changes in HDKB 2.2

The `EncryptionPHDSConfig` constructor has been enhanced to allow a Key Server certificate to be specified. The Key Server certificate is required only when using remote key delivery with and embedded machine-bound or domain-bound license.

Note: The Key Server certificate is an Adobe-issued license server certificate. Your Adobe Access Key Server must be configured with the corresponding private key. For more information on setting up a Adobe Access Key Server, see the [Adobe Access 4.0 Key Server documentation](#).

The following code snippet shows the changes made to the `EncryptionPHDSConfig` class:

```
class EncryptionPHDSConfig
{
public:
    HDSExport EncryptionPHDSConfig(
        const std::vector<adbe::hds::ByteBuffer>& licenseServerCredentials,
        /* License server credentials */

        const std::vector<fms::String>& licenseServerCredentialPwds,
        /* Licenese server credential passwords */

        const std::vector<adbe::hds::ByteBuffer>& recipientCerts,
        /* Shared Domain Certificates */

        const std::vector<adbe::hds::ByteBuffer>& additionalLicenses,
        /* Additional licenses need to be embedded */

        const std::vector<adbe::hds::ByteBuffer>& swfIdentifiers,
        /* SWF identifier hashes */

        const std::vector<adbe::hds::AirIdentifier>& airIdentifiers,

        uint32_t maxTimeToVerifySWFDigest,

        bool generateChainedLicense = false,

        adbe::hds::ByteBuffer keyServerCertificate=adbe::hds::ByteBuffer(),
        /* Specify Key Server certificate to support embedded (non-chained)
        license with Remote Key Delivery */

        fms::String objectID="");

    /* New getter method */
    dme::Opaque keyServerCertificate();
}
```

Chained License Generation

This feature supports embedding a leaf license with the content. In this case, the client is only required to acquire the root license from the license server in order to consume the protected content. For more information, see *License Chaining* in [Using Adobe Access Server for Protecting Content](#).

To support this workflow, the application must provide PHDS parameters without the recipient certificate and set `EncryptionPHDSConfig::generateChainedLicense` to true. The policy used to protect the content must have “Enhanced License Chaining” enabled and a root encryption key assigned. For more information, see *Enhanced License Chaining* in [Using Adobe Access Server for Protecting Content](#).

Fragmentation

If `setConfiguration` is not called before starting the fragmentation process, a default object of `FragmenterConfig` is used. The default object disables content encryption and writes the data from the messages into the file without any encryption.

If the application intends to create F4F files, it has to first write the `ftyp` box into the fragment file. The `ftyp` box is usually followed by a `moov` box. Fragments follow the `moov` box in an F4F file. Some data in the `moov` box needs to be updated as more fragments are added to the segment. To enable this, the SDK expects the user to pass an object that implements the `IMoovBuffer` interface. The SDK makes a call to the object implementing `IMoovBuffer` to get the size of the `moov` buffer for that fragment. The SDK then updates the contents of the `moov` box for the segment as samples are added to that segment. If the application is not interested in the `moov` box (because it is not creating F4F files), the application can skip calling the `setMoovBuffer` function. For more information, see the

`IMoovBuffer` interface.

The application can start streaming by adding messages (which will be added into the “current” Fragment) using `pushMessage` or `pushMultipleMessages` function. When the application determines that the current fragment is complete, `IFragmenter::completeFragment` has to be called. The function returns, through parameters, a pointer to the fragment built and a `FragmentContext` object that has information about the fragment and also any updates to the previous fragments. Also, this will update the moov buffer with relevant information.

After calling the function `completeFragment`, the `Fragmenter` ends the current fragment and starts a new fragment. The application can start adding samples into the new fragment by calling the `pushMessage` or `pushMultipleMessages` function. All of these samples will be added to the new fragment.

As a developer, you need to understand the basic requirements for fragment creation. Remember the following points:

- The first sample of a video track must be a key frame in every fragment. If the first video sample in a fragment being created is not a key frame, the `pushMessage` function fails and returns an error `eFirstVideoSampleNotKeyFrame`.
- If the media uses the H.264 video codec, all the AVCC messages corresponding to the decoder configuration must be sent before sending any samples. Similarly, if the media uses the AAC or HEAAC audio codec, the decoder configuration messages for the codec must be sent before any sample in the audio track.
- All the codec configuration messages must be sent before terminating the first fragment. No new codec configuration messages can be sent after the first fragment is created. These messages are important for H.264 video and AAC audio because once the first fragment is created a commitment is provided to the size of the set of Sample Descriptors (SDs). Those entries must not grow once the first fragment has been committed.

The above rules ensure that a proper size for the moov box is determined before the fragments are written. Because the fragments are written immediately after the moov box in an F4F file, the size of the moov box must be determined before writing any fragment. These restrictions apply even if the application does not intend to create F4F files and therefore does not set the moov buffer using the `setMoovBuffer` function.

If a gap needs to be inserted into the fragment run-table, `addSkip` method can be used. This function can be used to introduce gaps in fragment numbers or timestamps, or both. Pass a structure of type `SkipInfo` that has information about the type and data about the gap, as an argument. Gaps also signify a temporary absence of content.

When the application determines that a new segment should be started, it can call the `completeSegment` function. Internally, the fragmenter terminates the current segment. Further calls to `pushMessage` or `pushMultipleMessages` functions cause the samples to be added to the

first fragment in the next segment. Since this function is called to terminate the current segment, the moov buffer is reset and the application has to call the function `setMoovBuffer` again to set a new moov buffer (if the application is writing the fragments into F4F files, it has to open a new file and create an object that implements the `IMoovBuffer` interface and it has to write to the file). For a sample implementation, refer to the sample `offlinepackager` code supplied along with the SDK.

When the stream reaches completion, the application can call `notifyEndOfStream` to indicate to the SDK that no more samples are to be added. After calling `completeSegment` or `notifyEndOfStream`, samples should not be written to a fragment. If there are any samples pending, these two functions fail and return an `ePendingMessages` error. The application must call either `completeFragment` or `abort` after calling `pushMessage` and before calling `completeSegment` or `notifyEndOfStream`. The `abort` function will abort the fragment currently in process, and discard all the messages. The application can call this if it wants to discard the samples pushed so far.

The `IFragmenter` interface does not create the manifest (F4M) files or the index files. These files are created by the application. However, the SDK has support functions you can use to query for the data required to create the manifest files. The samples include code to generate manifest and index files.

The `getMetadata` function returns metadata in a `ByteBuffer` object that can be used to create an `OnMetadata` message. When encryption is used, the application can use the static function `getDRMAdditionalHeader` from the `IEncryptor` interface to get the string that specifies the metadata for the DRM stream. See the *Adobe HTTP Dynamic Streaming SDK Broadcast Sample Applications* document for information on how to generate manifest and index files.

There may be cases when an application creating fragments out of a live stream using the SDK, fails and needs to be restarted. The SDK supports a restart scenario for this. In such cases, the application on restart should set the flag `kFMRestart` when creating the fragmenter. This flag can be set only along with `kFMLive`. After the `Fragmenter` object is created, `setConfiguration` must be called with the same values as in the original run. After that, to make the `Fragmenter` initialize to the same state, the API `rebuildState` must be used. This API takes the last complete fragment from the crashed run and its number as input. This fragment must have at least one sample from each of the tracks present in the source media. Optionally, the metadata and bootstrap information can also be passed.

Once the call `rebuildState` completes, the application can call the function `addSkip` for any fragments that were missed. After that, the application can start adding samples again using the `pushMessage` or `pushMultipleMessages` API. If the application is writing the fragments into an F4F file, a call to `setMoovBuffer` must be made with an object that overwrites the previous `MoovBuffer`. If the configuration is exactly the same as the configuration at the initial time, the `MoovBuffer` will be the same size as the initial configuration. However, the segment duration for the current segment (in the `FragmentContext` structure returned by `completeFragment`) does

not get updated correctly after the restart for the interrupted segment. It is the responsibility of the application to update the index (F4X) file and remove any partial fragment from the fragment (F4F) file.

IEncryptor

The `IEncryptor` interface is primarily a fragment-level interface as opposed to a message-level interface of `IFragmenter`. It can be used to process and encrypt fragment objects that have been created in memory - either using `IFragmenter`, or using ones that have been constructed using the `IFragmentExtractor`. To support use cases that require creation of an F4F file with encrypted fragments, `IEncryptor` also provides file-level APIs (such as `setMoovBuffer`) similar to `IFragmenter`. `SetConfiguration` function is used in a similar way to set the fragmentation parameters and the encryption parameters. For details about the `EncryptorConfig` and `EncryptionConfig` structures, refer to the doxygen documentation included in the SDK.

The primary functions of the `IEncryptor` interface are `setEncryptorConfig` and `encryptFragment`. The API `setEncryptorConfig` is used to set the parameters for content protection, for example, the key to be used to encrypt content, the license server, and other required metadata. The `encryptFragment` function takes a smart pointer to an unencrypted fragment, creates a fragment with protected content, and returns a smart pointer to the fragment. This `IEncryptor` interface can also be used to generate F4F files with single or multiple fragments. To support that, `IEncryptor` has `setMoovBuffer`, `completeSegment`, and `completeFragment` interfaces, which function in the same way as in the `IFragmenter` interface.

The static function `getDRMMetadata` returns the DRM metadata in a `ByteBuffer` object. The `getDRMAdditionalHeader` function returns the base-64 encoded version of the DRM related information that can be directly written as a `drmAdditionalHeader` element in the manifest file.

`IEncryptor` also supports Object IDs like other objects of the SDK. An object ID can be passed to the `createEncryptor` function. This object ID is set as the property of the `Encryptor` object and is passed to all the logger callbacks arising from the `Encryptor` object.

`IEncryptor` has a new method `encryptMessage`, which takes one `MediaMessage` object and encrypts it. It should be used only when the user wants to segregate the steps of encrypting content from the actual packaging of the content into a fragment. The users of the new method will have their own utility libraries to package the encrypted content into a fragment.

This is an alternate to using the `encryptFragment` method of `IEncryptor`. Using the new method, the user can save the overhead of unpacking and repacking of the fragment incurred by the `encryptFragment` method.

All the messages except the audio and video sequence headers should be encrypted using the new method.

To use the `encryptMessage` method introduced in HDKB 2.2, you need to invoke the `setEncryptorConfig` method as usual, and then pass the media messages to be encrypted

through the `encryptMessage` method. The DRM metadata can be obtained by using `IEncryptor::getDRMAdditionalHeader` method. Methods like `setMoovBuffer`, `completeSegment`, and `completeFragment` will have no effect while using the `encryptMessage` method, as they deal with fragment at creation.

Instead of creating a HDS fragment and calling the `encryptFragment` method, you can call the `encryptMessage` method on every message and then push those messages to `Fragmenter` by calling the `pushMediaMessage` method. This will avoid the overhead of unpacking and repacking incurred by the `encryptFragment` method. Note that every message in the stream must be passed through this method and the encryption should be disabled in the `FragmenterConfig` on the `IFragmenter` object to which the encrypted messages are passed (since the messages are already encrypted).

Also, `IEncryptor` has a new method `getEncryptedRK` to support HLS content that uses key rotation:

```
class IEncryptor : public IRefCountedObject
{
    public:

        /// Encrypts a given MediaMessage.
        /// The encryption config is supplied by
        /// call to setEncryptorConfig
        /// Audio and Video Sequence headers shouldn't be encrypted.
        /// \param[in] inMsg Reference to an Unencrypted message.
        /// \param[out] outEncryptedMsg Reference to an Encrypted
        /// message.
        /// \return Success or an error code indicating the error.
        virtual ErrorCode encryptMessage
            (MediaMessage& inMsg, MediaMessage& outEncryptedMsg);

        /// Returns an encrypted rotation key for use with
        /// Adobe Access protected M3U8 with key rotation.
        ///
        /// \param[in] config EncryptionConfig required
        /// for drmMetaData, specifies license key.
        ///
        /// \param[in] rotationKey Key used to encrypt HLS segment.
        ///
        /// \param[out] encryptedRK Rotation Key
        /// encrypted with license key.
        ///
        /// \param[in] objId string to identify the object
        /// or the function call.
        static ErrorCode HDSExport getEncryptedRK(
            const EncryptionConfig& config,
            ByteBuffer& rotationKey,
            ByteBuffer& encryptedRK,
            const std::string& objId = "");
}
```

IFragmentExtractor

This interface is useful for processing an existing fragment (F4F) file. It has one primary function `getNextFragment`, which iterates over an F4F file, creates an in-memory representation of each fragment and returns a smart pointer to the next fragment object on each call.

This is useful, for instance, to create a new fragmented media (F4F, F4M, and F4X) file with protected content from unencrypted and fragmented media. This particular use case is implemented by the *EncryptorApp* sample application.

IMediaSource, IFileSource, ISampleIterator, and IMsgInfoIterator

The `IMediaSource` interface is used by the sample application to extract samples to be passed to the `Fragmenter` to create fragmented media. One implementation that is provided with the SDK is a utility to parse and create an `IMediaSource` from an F4V file. The API `createF4VFileSource` can be used to construct an `IMediaSource` reference from an F4V or an MPEG4 media file, which can be used to extract `MediaMessage` objects to be given as input to the SDK for fragmenting. An example of this usage is in the *OfflinePackager* sample application. From version 1.1, the `getSampleIterator` API of `IMediaSource` can take two input parameters, `startTime` and `endTime`, indicating to the `MessageIterator` to parse the messages only between the specified timestamps.

From version 1.1, `IMediaSource` also provides an API to iterate over all the `MessageInfo` objects. `MessageInfo` objects can be used to get basic information about the messages contained in the file except the actual video/audio data. This can be useful for some cases, for instance, in determining the fragment boundaries without actually fragmenting the content. From version 1.1, the SDK can process encrypted F4V files (created using the Adobe Access SDK).

The API `createF4VFileSource` can handle encrypted F4V (or MPEG4) files, and the `IFragmenter` can receive encrypted messages to fragment. The `createF4VFileSource` API also takes an object ID to be set on the `MediaSource` object created. This object ID will be set as the identifier of the `MediaSource` object and is passed to the `logMessage` call from this object.

IFragment

The `IFragment` interface represents a fragment of media stream in memory. It implements the following functions:

- `serialize` – This function helps in extracting the messages from a fragment. This function is useful to persist the fragments either in an F4F file or in any other format. The pointer passed to it can be a memory buffer or can be a memory-mapped file. As a developer, you need to ensure that the required amount of memory is allocated to the pointer, which is passed as an argument. To know the size of the serialized fragment, the application can call the function `getSerializeLength`. Before calling the `serialize` function, the application must call the `setBaseOffset` function with the offset in the file where this fragment will get written. This is because a fragment has fields that need to have the actual file offset.

- `getBootstrapInfo` and `getBootstrapLength` – These functions can be used to obtain the bootstrap information of a fragment. The application must first call `getBootstrapLength` to determine the size of the bootstrap info, allocate at least the required memory, and pass that pointer as an argument to the `getBootstrapInfo` function. The `getBootstrapInfo` function does not perform any check on the pointer before attempting to write to the location defined by the pointer.
- `getSampleIterator` – This function will return an object that implements the `IMsgIterator` interface, which can be used to extract all the samples in the fragment as `MediaMessage` objects.

MediaMessage

This class represents each message containing the samples, as well as other messages, such as `OnMetaData` and the codec header messages. Each message holds a smart pointer (`IMemoryArenaAllocPtr`), which in turn holds a reference to the data. In all the messages that are created by the SDK, this is a smart pointer object and only the reference is copied and not the data. Therefore, copy or pass-by-value of `MediaMessage` objects is not expensive, and the data gets cleaned up automatically when all the `MediaMessage` objects referencing this data go out of scope. For an example of creating `MediaMessage` objects, refer to the `FlvMediaSource` class provided as part of the samples.

IMsgIterator

This class presents basic information about messages without the actual message contents such as video, audio, or data. This class can be useful for various purposes like identifying the fragment boundaries without actually going through fragmentation of content. These objects are much smaller in comparison to `MediaMessage` objects and therefore are more efficient to iterate through and process.

IMoovBuffer

The `IMoovBuffer` interface has two functions:

- `init` – This function tells the object the maximum size of the buffer. The SDK calls this function before writing any data into the buffer. The application can use this function to reserve the specified space in the file for the moov box and write the fragments after that offset in the file.
- `write` – The SDK calls this function whenever it wants to write some data (for example, track information and movie duration) into the moov box. This function has an offset, data, and the size of the data to be written as parameters. The offset is the offset from the beginning of the buffer and not the beginning of the file.

For a sample implementation of `IMoovBuffer`, refer to the file `moovbuffer.cpp` in the samples supplied with the SDK.

ILogger and IMultiLogger

The `ILogger` and `IMultiLogger` APIs are callback interfaces. The SDK uses these interfaces to log diagnostic messages. Applications should implement one of these two interfaces and use the `setLogger` API to set an object that receives log messages as callbacks. This helps in receiving more descriptive messages, such as whenever an API returns an error.

The only difference between the `ILogger` and `IMultiLogger` interfaces is that `IMultiLogger` can receive an object ID (a string) with each log message. This will be useful when the SDK is being used in a multithreaded application or in a batch mode where multiple instances of each object are created and used to process different streams. In such cases, it is important to know the instance of an object from which the log message has come. For that purpose, one can set an object ID during the construction of each object, for example, the `Fragmenter`, and pass an object that implements the `IMultiLogger` API to the `setLogger` API. The `ILogger` API is the same except for the object ID, which means the callback functions of the `ILogger` interface do not receive an object ID with each log message and therefore may be hard to distinguish. If an application does not call the `setLogger` API, the SDK does not invoke the `logMessage` callback.

The level of detail of the log messages can be controlled by the `setLogLevel` API. There are five levels of logging supported. They are, in increasing order of details, `kLogNone`, `kLogError`, `kLogWarning`, `kLogInfo`, and `kLogDebug`. The default log level is `kLogWarning`. At this level, the SDK invokes the logging API for any warning or error messages. Setting the log level to `kLogNone` will completely stop logging. This is equivalent to not setting a logger at all.

KeyGenerator

`KeyGenerator` is a utility class available in the SDK. It can be used to generate random keys while using key rotation. The `KeyGenerator` class has the following functions:

- `initKeyGenerator` – This utility function initializes the `KeyGenerator` module. This function must be executed successfully before the user can use its random key generation facility. This function should only be called once (even in a multithreaded scenario).
- `generateKey` – This utility function generates a random key of the requested length (in bytes). On Windows, this uses the `CryptGenRandom` utility of the system to generate a random key. On Linux, it reads from “`/dev/urandom`” to generate the random key. The random key generation would have the same level of security as the corresponding system APIs used for key generation.
- `destroyKeyGenerator` – This utility function is called to de-initialize the `KeyGenerator` function. In order to use the `KeyGenerator` function again, you need to call the `initKeyGenerator` function.

Usage scenarios for HLS-based content protection

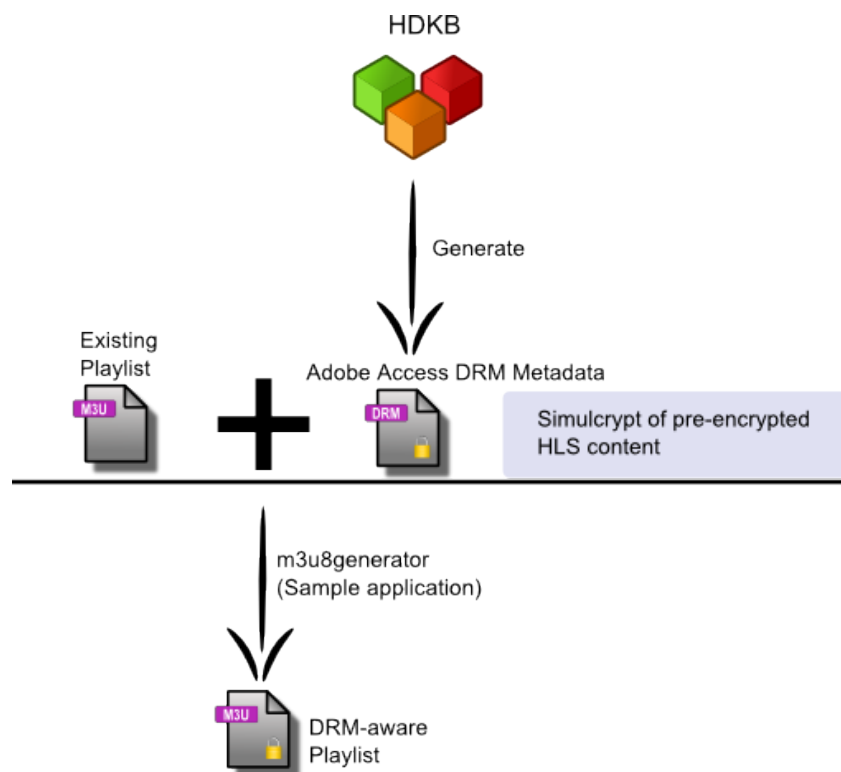
The following sections depict some common usage scenarios for HDKB APIs.

Making HLS content DRM-enabled

The HDKB APIs allow you to enable Adobe Access protection for pre-encrypted HLS content. You can take advantage of this feature if you want to utilize the Adobe Access for iOS APIs. Using the HDKB APIs, you can add Adobe Access information to the playlist, so it can be played by an Adobe Access iOS client. For instance, the `AccessPlayer` sample application shipped as part of the Adobe Access for iOS API bundle can play content protected by HDKB APIs through Adobe Access.

HDKB ships with a sample application, `m3U8generator`, which demonstrates the usage of the HDKB APIs to generate the DRM metadata, and then insert the metadata into the M3U8 file.

Figure 6: Generating DRM-aware playlists



See the *HDKB 3.1 Sample Applications Guide* for more information on the M3U8 Generator sample application.

Generating DRM metadata

You can use the HDKB APIs for generating DRM metadata. You generate DRM metadata by performing the following steps:

1. You construct an `EncryptionConfig` object, specifying the parameters required for metadata generation (and optionally license generation).
2. You invoke the static `getDRMMetadata` method of `IEncryptor`.

Based on the values passed in the `EncryptionConfig`, several types of DRM metadata can be generated supporting:

- Based upon embedding license in the metadata
 - Embedded License
 - No License server required (PHLS)
 - Chained license (Leaf license is embedded into the metadata, root license will be acquired from the License server)

- Non-embedded licenses (license will be acquired from the License server)
- Based upon key delivery
 - Local key delivery (No key server required)
 - Remote key delivery

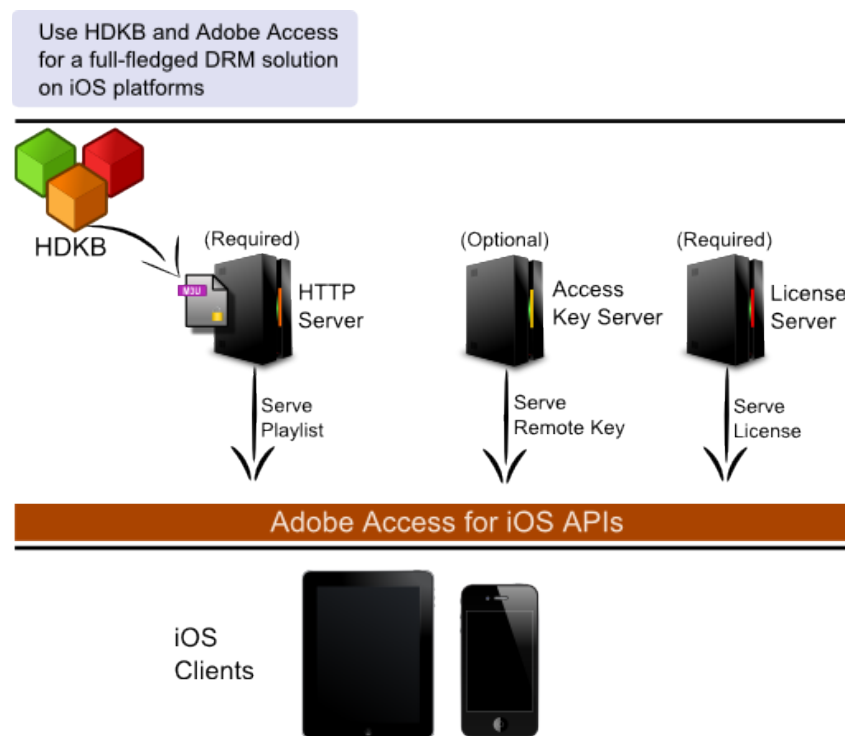
Note: For more information, see the [Adobe Access documentation](#).

Note: See the *HDKB 3.1 Sample Applications Guide* for examples on generating various supported metadata using the M3U8 Generator sample application.

Full DRM protection

You can use the HDKB APIs along with Adobe Access to build a full-fledged DRM solution for Apple iOS platforms. You can also use an Adobe Access Key Server for remote key delivery.

Figure 8: A full-fledged DRM solution



HDKB with PHLS

HDKB APIs can be used for Protected HLS streaming (PHLS), which uses the protection system to deliver the content license to the native iOS applications without requiring a license server.

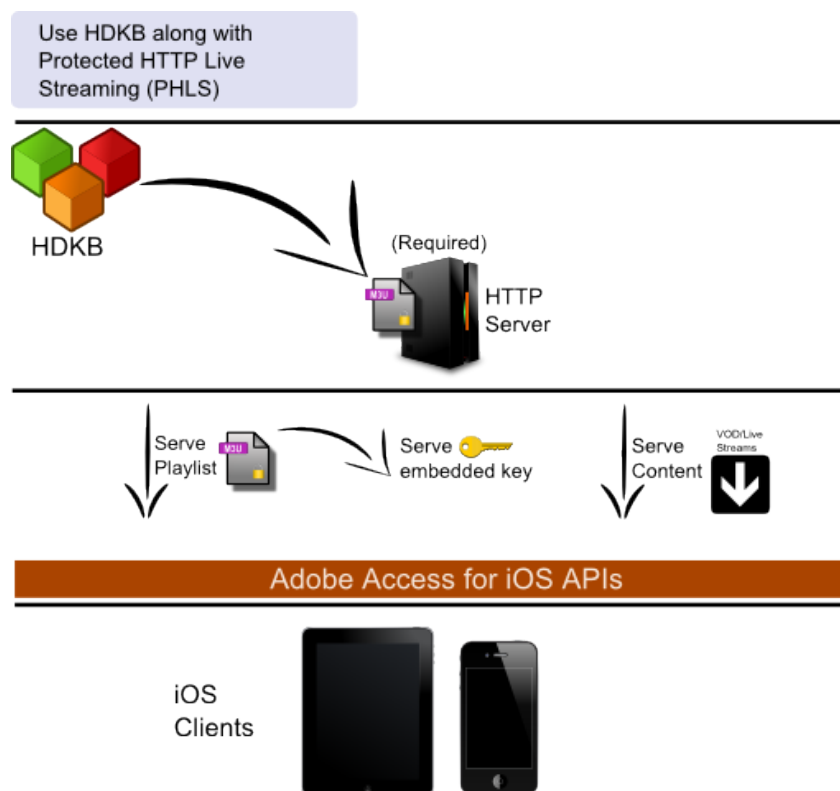
For HLS content, Adobe Access supports the following key delivery methods:

- Local
- Remote

The key delivery method is specified in the policy used in the metadata. If remote key delivery is selected and a license is embedded in the metadata, a key server certificate must also be provided in order to generate the license. Of the metadata variations discussed, key server certificate is only required for the “Adobe Access with embedded license” case. In order to support this use case with remote key delivery, `EncryptionPHDSConfig` is now enhanced to allow the key server certificate to be specified. See [EncryptionPHDSConfig changes in HDKB 2.2](#).

When HLS key rotation is used during content encryption, the key URL in the `EXT-X-KEY` tag in the Adobe Access M3U8 file must contain the `encryptedRK` parameter. The value of this parameter is the rotation key encrypted with the key in the Adobe Access license. In the case of PHLS, remote key delivery is not supported. The key is served from the embedded license available in the playlist.

Figure 9: HDKB for protected HLS



HDKB and Adobe Access can also be used in a DRM setup based on Adobe Access enhanced License chaining. With enhanced license chaining, the leaf license contains the Content Encryption Key (CEK), but instead of the license being bound to the device, it is bound to a root license. The root license is bound to the device. Since there is no device-specific information in the leaf license, it can be embedded in the content. To play the content, the user needs to obtain a root license from a license server, but a single root can be used to enable all the leaf licenses that were generated with the same policy.

Enhanced license chaining gives you a way to scale the license server, since the server does not have to issue a leaf license for every piece of content, since the leaf can be embedded without any loss of security. See *Using the Adobe Access 4.0 Server for Protecting Content* for more information on enhanced license chaining.

Generating an Adobe Access-enabled M3U8 file

An Adobe Access-enabled M3U8 file must contain the Adobe Access DRM metadata.

The metadata can be specified in one of the following ways:

- Base64-encoded Flash Access content metadata:
#EXT-X-FAXS-CM:MIUIISLSDKKDKDFJKJUMQYJKoZIhvcNAQcCoIIUIjCCFB4CAQEx...
- URI pointing to a file containing BER-encoded content metadata prefixed with "URI=". The URI may also be specified as a path relative to the location of the M3U8 file.
#EXT-X-FAXS-CM:URI="http://host:port/content/asjhg198267.metadata"

To obtain the DRM Metadata, perform the following steps:

1. Construct `EncryptionConfig` to specify the parameters required for metadata generation. See the *HDKB 3.1 Sample Applications Guide* for M3U8 Generator examples and sample parameters.
 - a. If key rotation was used during content encryption, randomly generate a new key to pass as the key parameter. In Adobe Access, this additional key is needed to encrypt the actual rotation keys within the Access system and is distinct from the keys used to encrypt the HLS content.
 - b. If key rotation was not used during content encryption, the key parameter must match the key used for content encryption.
2. Use `IEncryptor.getDRMMetadata()` to obtain the DRM metadata bytes.
 - a. Base-64 encode the value and embed it in the M3U8 file **OR**
 - b. Write the plain bytes to a file, and include the file URI in the M3U8 file.

Key URLs

Adobe Access requires modification of the URI in the `EXT-X-KEY` tag.

Local Key Delivery

For local key delivery, the URI must be `"faxes://faxes.adobe.com"`:

```
#EXT-X-KEY:METHOD=AES-128,URI="faxes://faxes.adobe.com"
```

Remote Key Delivery

For remote key delivery, the URI must be the URL of your Adobe Access Key Server:

```
#EXT-X-KEY:METHOD=AES-128,URI=https://host:port/faxsks/tenant/key
```

Key Rotation

When Key Rotation is used (with either local or remote key delivery), an `"EncryptedRK"` query parameter must be added to the URI. The value of this parameter is `"0x"` + the hex encoded rotation key:

```
#EXT-X-KEY:METHOD=AES-128,URI="faxes://faxes.adobe.com?EncryptedRK=0x00112233445566778899aabbccddeeff"
```

To generate the value of the `EncryptedRK` parameter:

- Use the same `EncryptionConfig` used to generate the DRM metadata.
- Use `IEncryptor.getEncryptedRK()` to obtain the encrypted rotation key, then hex-encode the value and prepend `"0x"`.

Copyright

© 2013 Adobe Systems Incorporated. All rights reserved.

Adobe HTTP Dynamic Streaming SDK Broadcast API Overview
Edition 3.5

This guide is licensed for use under the Creative Commons Attribution Non-Commercial 3.5 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.