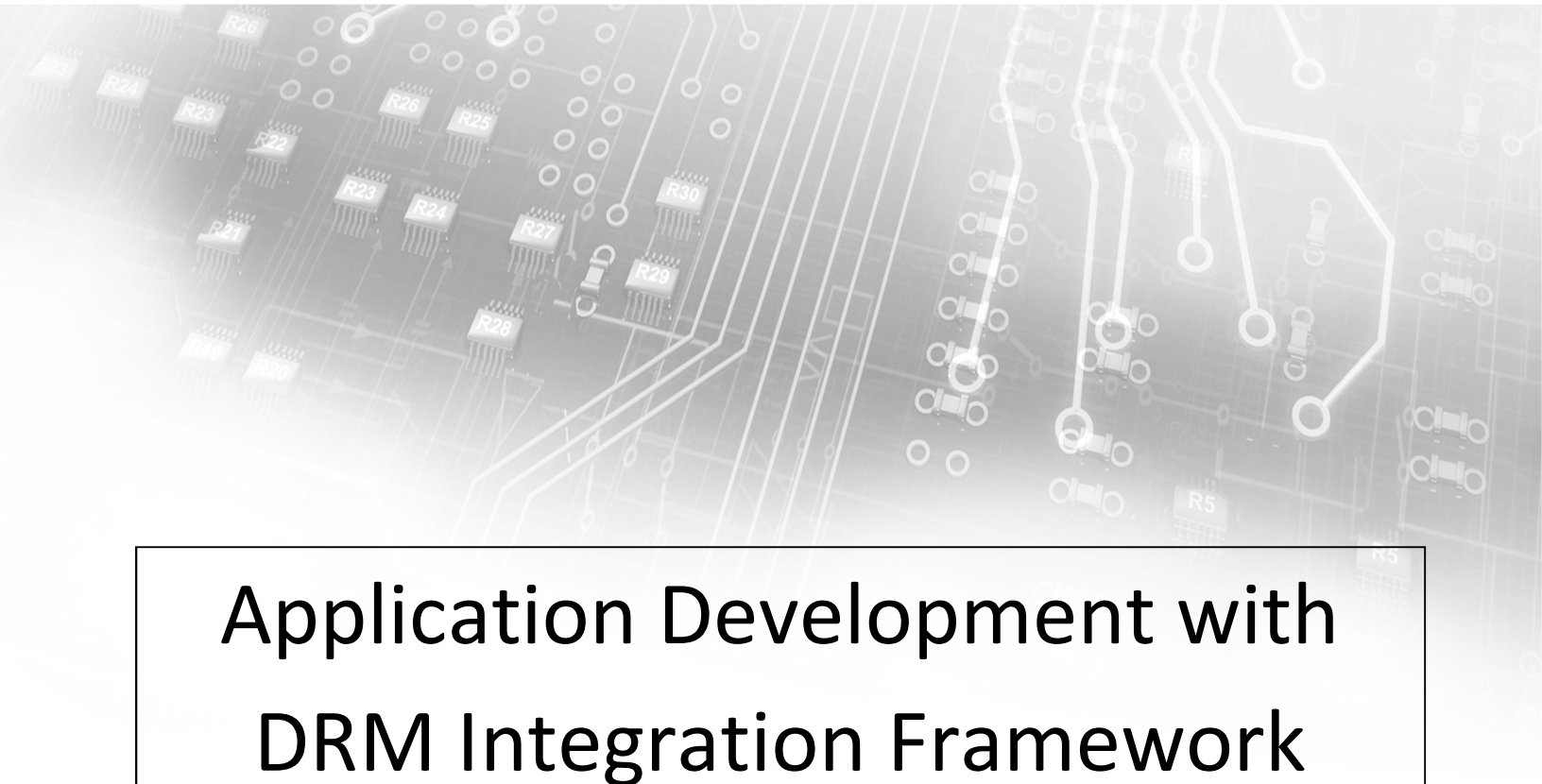


Application Development with DRM Integration Framework

Version 1.4

Broadcom
1320 Ridder Park Drive
San Jose, California 95131
broadcom.com

Table of Contents

1 Revision History

Issue	Date	By	Change
1.0	10/16/2015	Yuichi Aiba	First Release
1.1	12/4/2015	Yuichi Aiba	Minor updates in a couple of diagrams. Descriptions in “Set up Parser” and “Set up Decryptor” were modified.
1.2	12/21/2015	Yuichi Aiba	Modified CreateStreamer() API – no need to specify true/false
1.3	1/7/2016	Yuichi Aiba	Minor updates in diagrams and example code
1.4	10/3/2017	Yuichi Aiba	Streamer OpenPlaypump() API was modified so that users can specify NEXUS_PlaypumpOpenSettings

2 Reference

DRM Integration Framework User Guide – DIF_User_Guide.pdf

3 Glossary

Abbreviation	Description
CDM	Content Decryption Module
CENC	Common Encryption File Format
DRM	Digital Right Management
PES	Packetized Elementary Stream
PIFF	Protected Interoperable File Format
SAGE	Secure Applications Guardian Engine
SRAI	SAGE Remote Application Interface (SRAI)
SVP	Secure Video Path
TEE	Trusted Execution Environment, secure environment in which assets are protected, sensitive operations occur, etc.
URSR	Unified Reference Software Release

4 Background

Broadcom supports Secure Video Path (SVP) feature to protect decrypted data from users/apps accesses. However it is complicated for application developers to utilize/integrate the SVP feature into applications since the developers have to utilize API's across several different modules such as Common DRM, Widevine CDM, SRAI, Nexus DMA, etc. We integrated the SVP feature in several applications and frameworks, and have experienced duplicated steps and debugging difficulties.

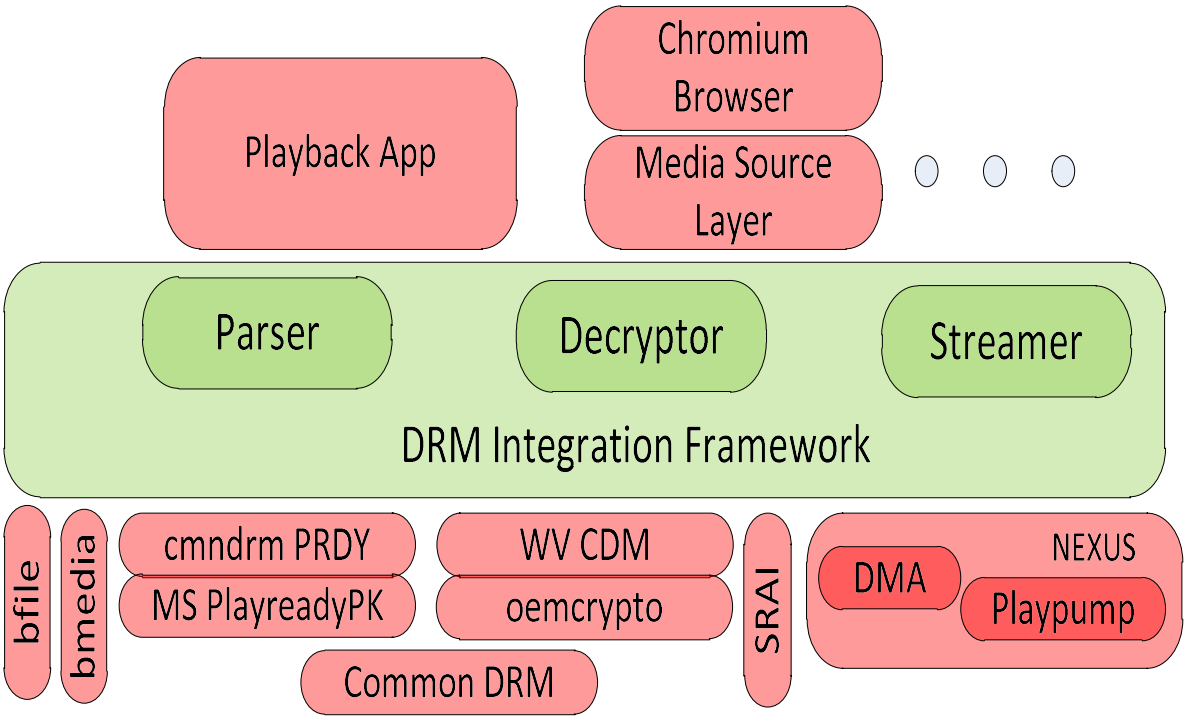
The goal to have this new DRM Integration Framework is to pack the common steps and routines into a separate module and provide application developers with easy-to-use API's to simplify their application development.

5 Overview

The DRM Integration Framework will provide the following features:

- Parser: reads audio/video mp4 data and retrieves information required for provisioning, acquiring licenses, and processing fragments and samples.
- Decryptor: provides functions for provisioning, acquiring licenses and decrypting MP4 sample data.
- Streamer: sets up playpumps, buffers and DMA transfer, and pushes data to AV pipeline.

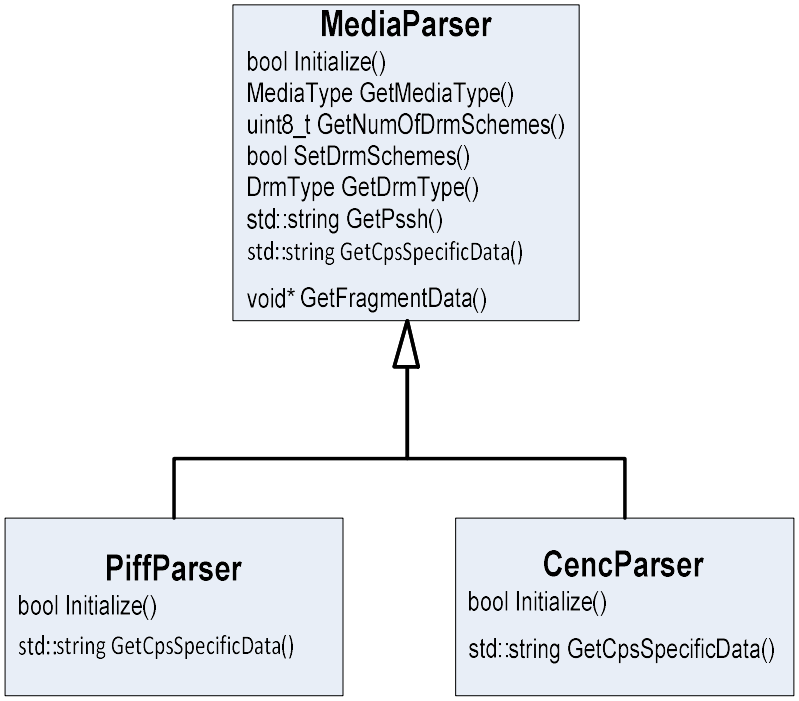
This diagram shows the position among related modules.



5.1 Parser

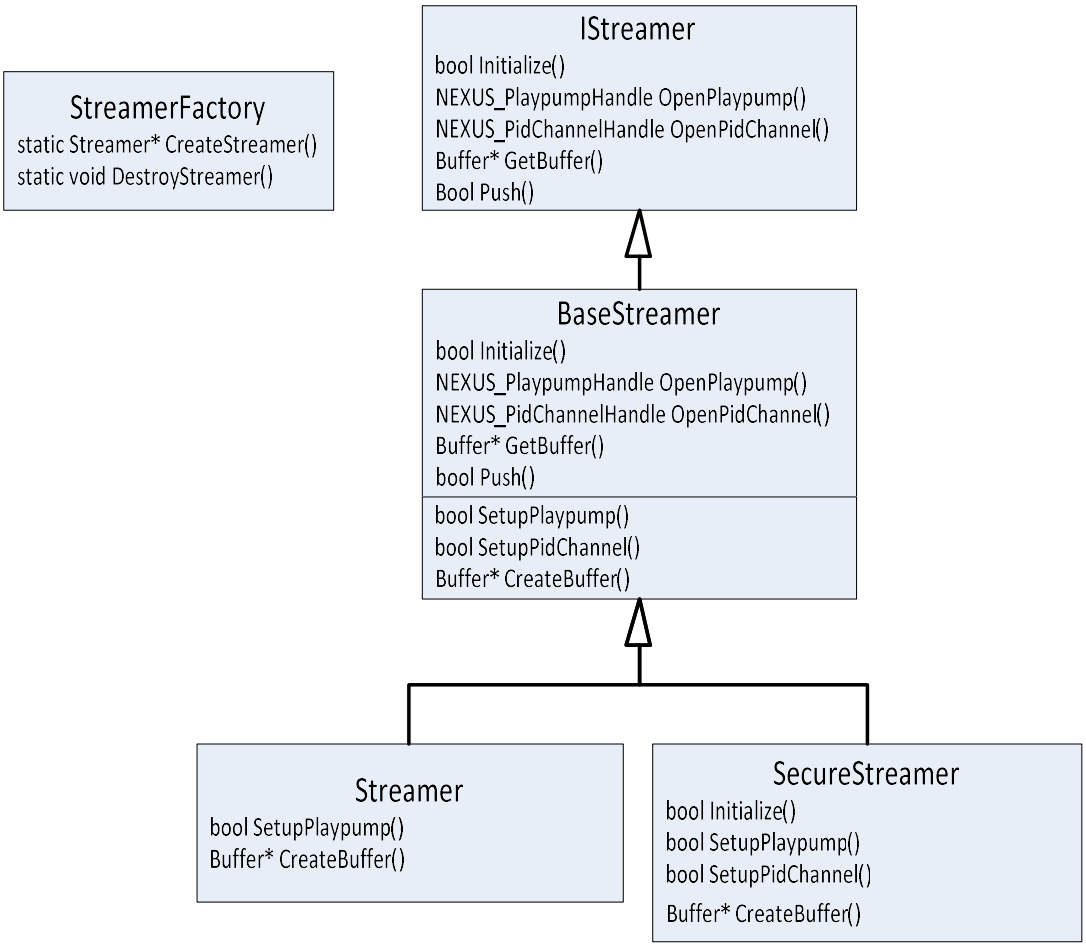
The Parser analyzes input steam and provides data required for decryption such as media type, DRM type, PSSH, Content Protection System Specific Data and content samples in an encrypted media file.

PIFF and CENC are the two container types supported currently. Please note that a single media file may contain two or more DRM schemes inside. The Parser scans all DRM schemes for user to choose and picks the right one at user's choice.



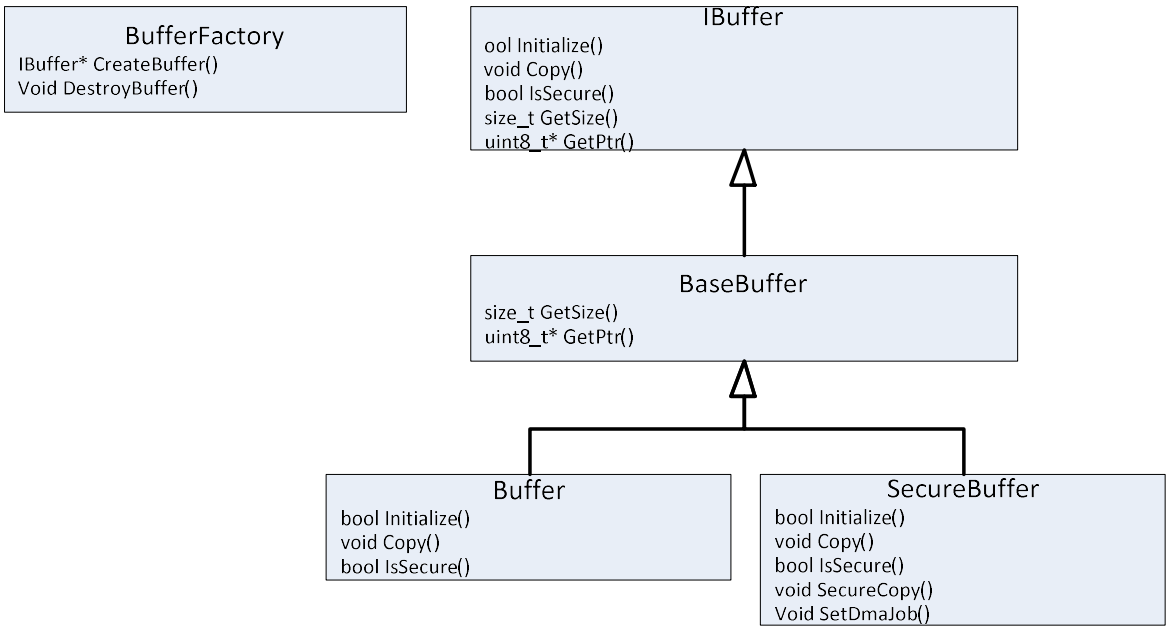
5.2 Streamer

Streamer is used to push data to audio/video pipeline. Users can create either a SecureStreamer for SVP or a Streamer for non-SVP through StreamerFactory interface. Users can open one Nexus playpump using a Streamer and can't open more than one playpump per Streamer.



5.3 Buffer

Buffer is a helper class to handle header information and encrypted/decrypted data between Streamer and Decryptor. SecureBuffer is used together with SecureStreamer, but usually users don't have to know whether the Buffer is secure or not.

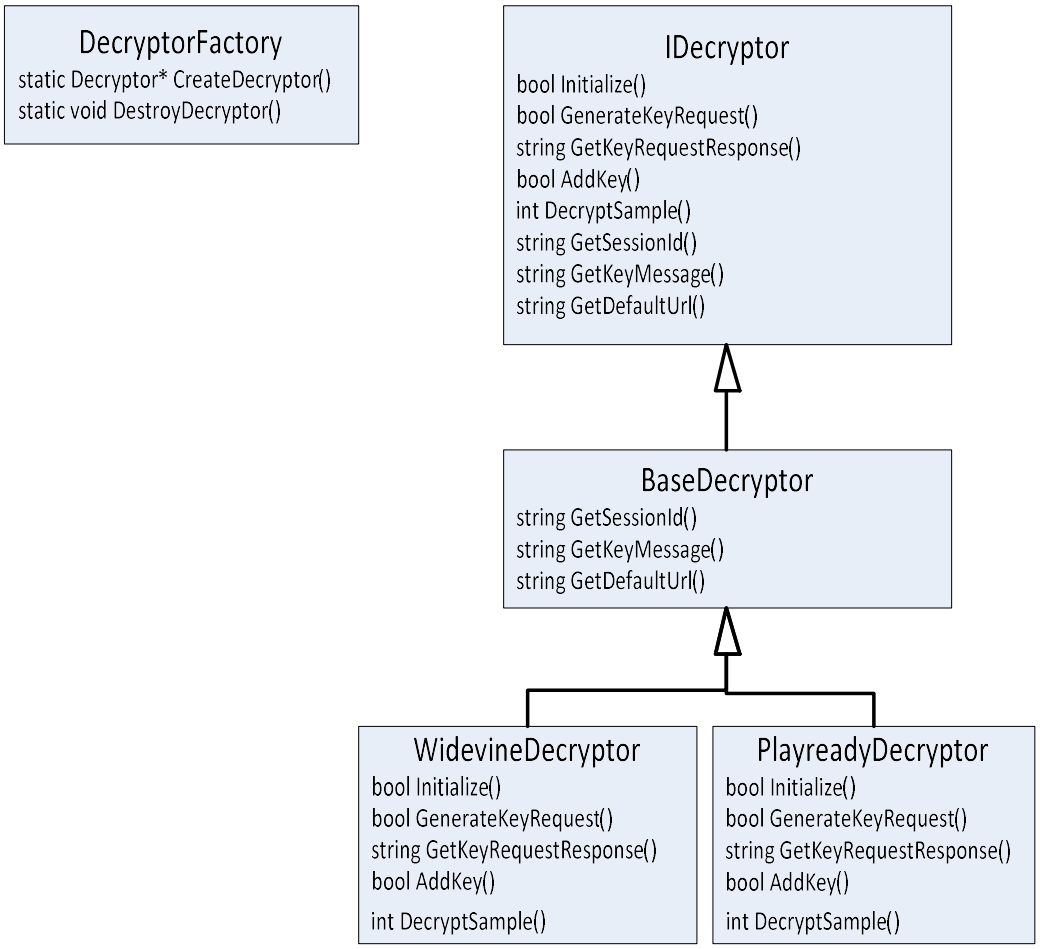


5.4 Decryptor

Decryptor takes these roles:

- Setting up underlying security modules
- Device provisioning including communication with the provisioning server (Widevine)
- Generating key request to send to a license server
- Communicating with a license server to acquire a license
- Adding the acquired license
- Decrypting each frame

We provide Widevine and Playready Decryptors.



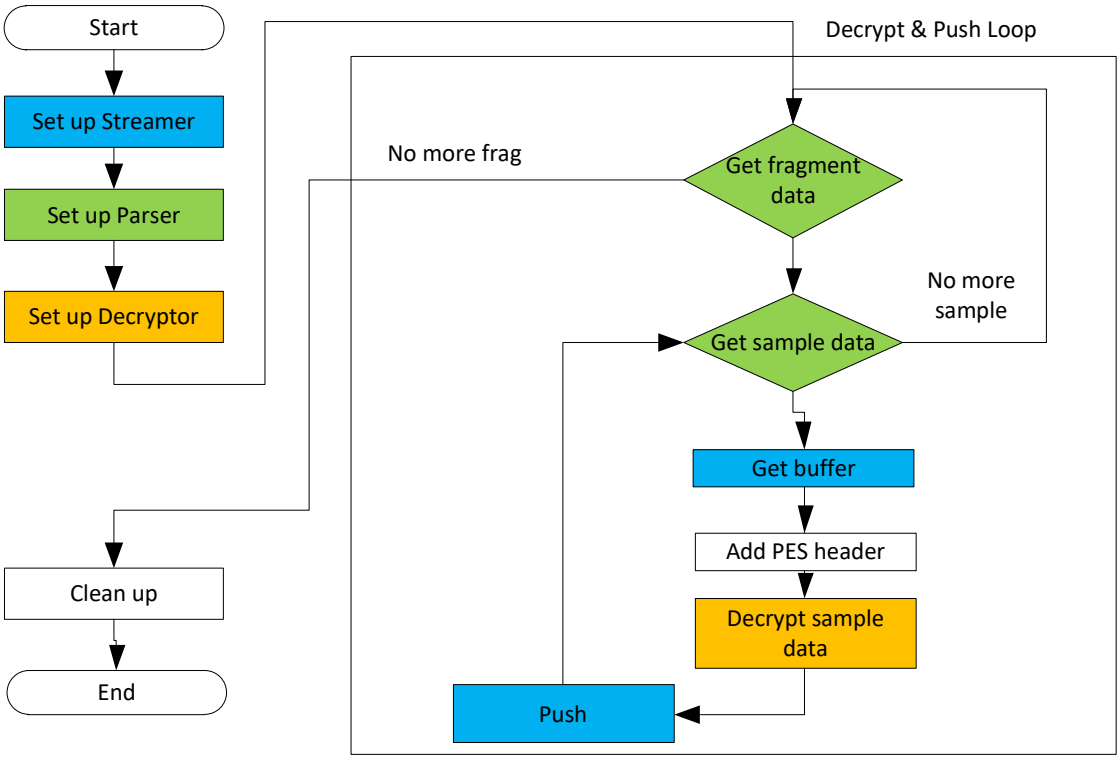
6 Application Development Guideline

6.1 Basic Application Workflow

The below diagram shows an example of application workflow. An application should have 4 main steps:

- Streamer set-ups: Application will usually set up two Streamers, one for audio and another for video.
- Parser set-ups: Application will feed mp4 files to Parser, get DRM information in the files and select DRM type to use for decryption.
- Decryptor set-ups: Application will create a Decryptor and initialize it by handing some DRM information from Parser.

- Decrypt & Push loop: This is the key phase of the application. The application will get data from Parser, feed it to Decryptor to decrypt and push it to Streamer. This is done in a loop to process all fragment data in the MP4.



The following sections will describe each block further.

6.2 Create Streamer objects and open playpumps

This example code shows how to create Streamer objects for SVP case. Two playpumps are opened through two Streamer objects, both of which are configured with secure playpump buffers in this example. The application can open playpumps through Streamer's API and configure them with the own callbacks, codecs, etc, but the application has to get pid channels through Streamer's OpenPidChannel() API.

```
    // create Streamer
    IStreamer* videoStreamer = StreamerFactory::CreateStreamer();
    IStreamer* audioStreamer = StreamerFactory::CreateStreamer();
...
    // get default playpump open settings and modify the settings
    videoStreamer->GetDefaultPlaypumpOpenSettings(&playpumpOpenSettings);
    playpumpOpenSettings.fifoSize = VIDEO_PLAYPUMP_BUF_SIZE;
    // open video play pump
```

```
    videoPlaypump = videoStreamer->OpenPlaypump(playpumpOpenSettings);

    // open audio playpump with default settings
    audioPlaypump = audioStreamer->OpenPlaypump(NULL);

    // you can configure the playpump
    NEXUS_Playpump_GetSettings(s_app.videoPlaypump, &playpumpSettings);
    playpumpSettings.dataCallback.callback = play_callback;
    playpumpSettings.dataCallback.context = s_app.event;
    playpumpSettings.transportType = NEXUS_TransportType_eMpeg2Pes;
    NEXUS_Playpump_SetSettings(s_app.videoPlaypump, &playpumpSettings);
...

    // get the PidChannel
    videoPidChannel = videoStreamer->OpenPidChannel(REPACK_VIDEO_PES_ID,
&video_pid_settings);
        audioPidChannel = audioStreamer->OpenPidChannel(REPACK_AUDIO_PES_ID,
&audio_pid_settings);

    // PidChannel is used to start the decoder
    videoProgram.settings.pidChannel = s_app.videoPidChannel;
    NEXUS_SimpleVideoDecoder_Start(videoDecoder, &videoProgram);
...
```

6.3 Set up Parser

This example shows how to set up a parser for a CENC stream. The application calls fopen() and then provides the file handle to create a Parser.

CENC may include multiple DRM schemes (Playready, Widevine, ClearKey, etc.). The application can set one DRM scheme to the Parser.

```
    FILE* fp_mp4 = fopen(mp4_file, "rb");

    // creating Cenc Parser
    MediaParser* parser = new CencParser(fp_mp4);

    // Initialize the Parser - it may fail if format is not CENC
    ret = parser->Initialize();
...
    // Get number of DRM types that are included in the MP4
    DrmType drmTypes[BMP4_MAX_DRM_SCHEMES];
    uint8_t numOfDrmSchemes = parser->GetNumOfDrmSchemes(drmTypes,
BMP4_MAX_DRM_SCHEMES);

    // Select one DRM type
    for (int i = 0; i<numOfDrmSchemes; i++){
        if (drmTypes[i] == requestedDrmType){
            parser-> SetDrmSchemes(i);
        }
    }
```

6.4 Set up Decryptor

This example shows how to set up a Decryptor. Three parameters used in this example (drmType, psshDataStr and cpsSpecificData) are acquired through Parser's API.

The application has to specify an appropriate license server URL for the mp4 file to request a license when calling GetKeyRequestResponse().

```
    // Get DRM type using parser
    drmType = parser->GetDrmType();

    // creating Decryptor using the drmType
    IDecryptor* decryptor = DecryptorFactory::CreateDecryptor(drmType);
...
    // Read Pssh: used to initialize the Decryptor
    psshDataStr = parser->GetPssh();

    // Initialize call includes provisioning
    decryptor->Initialize(psshDataStr)
...
    // Get Content Protection System Specific Data: used to generate a key request
    cpsSpecificData = parser->GetCpsSpecificData();

    // Generate key request to acquire a license from a server
    decryptor->GenerateKeyRequest(cpsSpecificData)
...
    // Request a license to a server
    std::string response = decryptor->GetKeyRequestResponse(license_server);
...
    // Add the received license
    decryptor->AddKey(response)

...
```

6.5 Decrypt and Push sample data

6.5.1 Get a sample data

The application can get sample information through parser and go on to process each sample data.

```
    for (;;) {
        // Read sample information
        decoder_data = parser->GetFragmentData(frag_info, app.pPayload,
decoder_len);
        // No more samples > exit from the loop
        if (decoder_data == NULL)
            break;

        // decoder_data is used to read AVCC/AAC config
```

```
        bmedia_read_h264_meta(&meta, decoder_data, decoder_len);
...
```

6.5.2 Handle PES header

This step has to be done at the beginning of handling each sample data. It is straightforward – users just need to get a Buffer through Streamer's GetBuffer API and copy the PES header to it. The application can destroy the Buffer right after copy is done.

```
    // create PES header
    pes_header_len = bmedia_pes_header_init(app->pVideoHeaderBuf, sampleSize,
&pes_info);

    // get buffer from the Streamer
    IBuffer* output = streamer->GetBuffer(pes_header_len);

    // copy PES header to the buffer
    output->Copy(0, app->pVideoHeaderBuf, pes_header_len);

    // you can destroy the buffer right after you use it
    BufferFactory::DestroyBuffer(output);
    bytes_processed += pes_header_len;
```

6.5.3 Clear data

If the sample data is clear, this step is just to copy the clear data to the Buffer.

```
    // get buffer from the Streamer
    IBuffer* output = streamer->GetBuffer(sampleSize);

    // copy clear data to the buffer
    output->Copy(0, sampleData, sampleSize);
    BufferFactory::DestroyBuffer(output);
    bytes_processed += sampleSize;
```

6.5.4 Encrypted data

If the sample data is encrypted, users have to prepare two Buffers – one for input encrypted data and the other for output decrypted data. Decryptor's DecryptSample API will take care of subsamples and decrypt the data into the output buffer. The application can destroy the Buffers right after DecryptSample is done.

```
    // set up input buffer for decrypt (encrypted data)
    IBuffer* input = BufferFactory::CreateBuffer(sampleSize, sampleData);
...
    // get buffer from the Streamer (this is going to be output of the decrypt)
    IBuffer* decOutput = streamer->GetBuffer(sampleSize);

    // call Decryptor's DecryptSample
    numOfByteDecrypted = decryptor->DecryptSample(pSample, input, decOutput,
sampleSize);

    // destroy buffers after decrypt
```

```
    BufferFactory::DestroyBuffer(input);
    BufferFactory::DestroyBuffer(decOutput);

    bytes_processed += numOfByteDecrypted;
```

6.5.5 Push data

After data are ready in Streamer, the application can just call Push to send data to AV pipelines.

```
...

    videoStreamer->Push(bytes_processed);
```

6.6 Clean up

At the end of the application, Decryptor and Streamers have to be destroyed using the factory classes.

```
        DecryptorFactory::DestroyDecryptor(decryptor);

        StreamerFactory::DestroyStreamer(videoStreamer);

        StreamerFactory::DestroyStreamer(audioStreamer);

        delete parser;
```