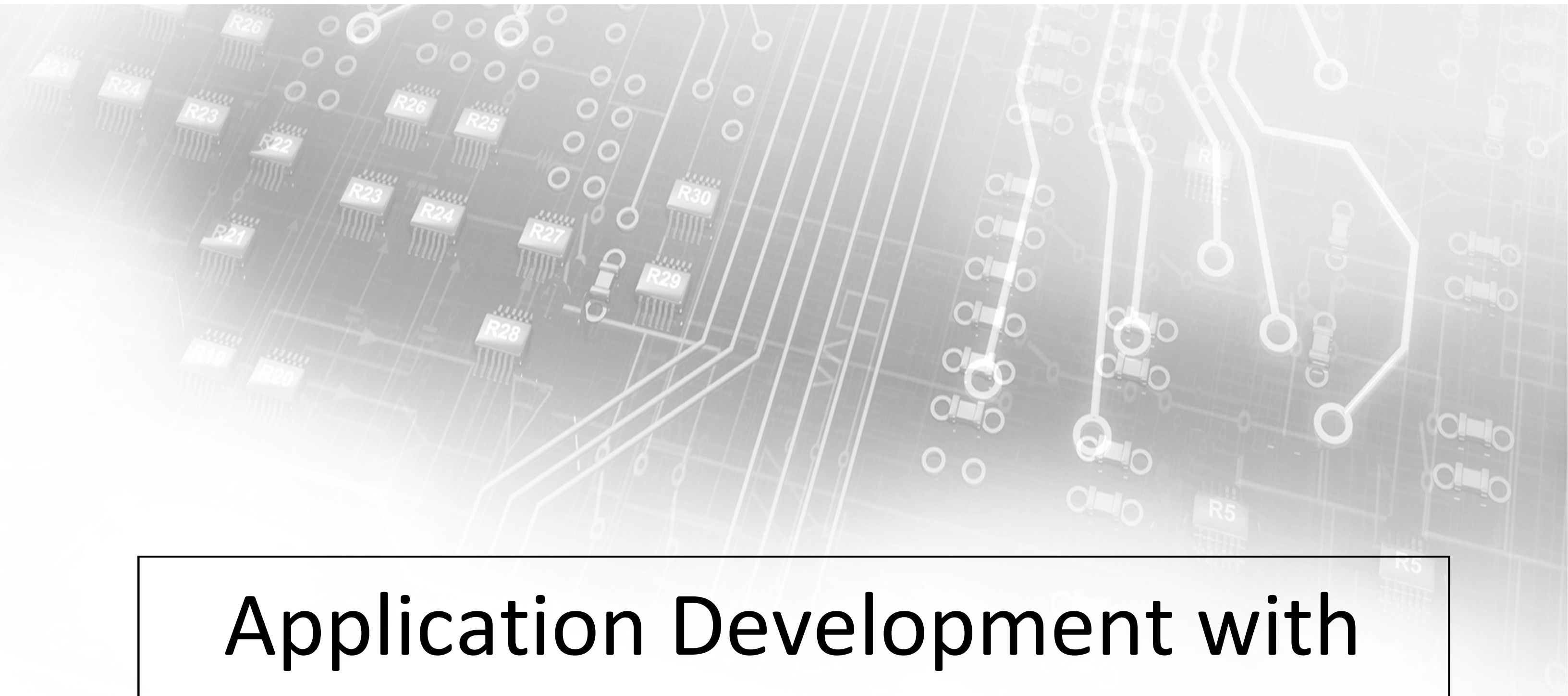


Application Development with DRM Integration Framework

Version 1.6

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
1.5	4/2/2018	Yuichi Aiba	GetSettings() and SetSettings() were added to Streamer
1.6	4/17/2018	Yuichi Aiba	Scatter-gather feature was added to Streamer

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.

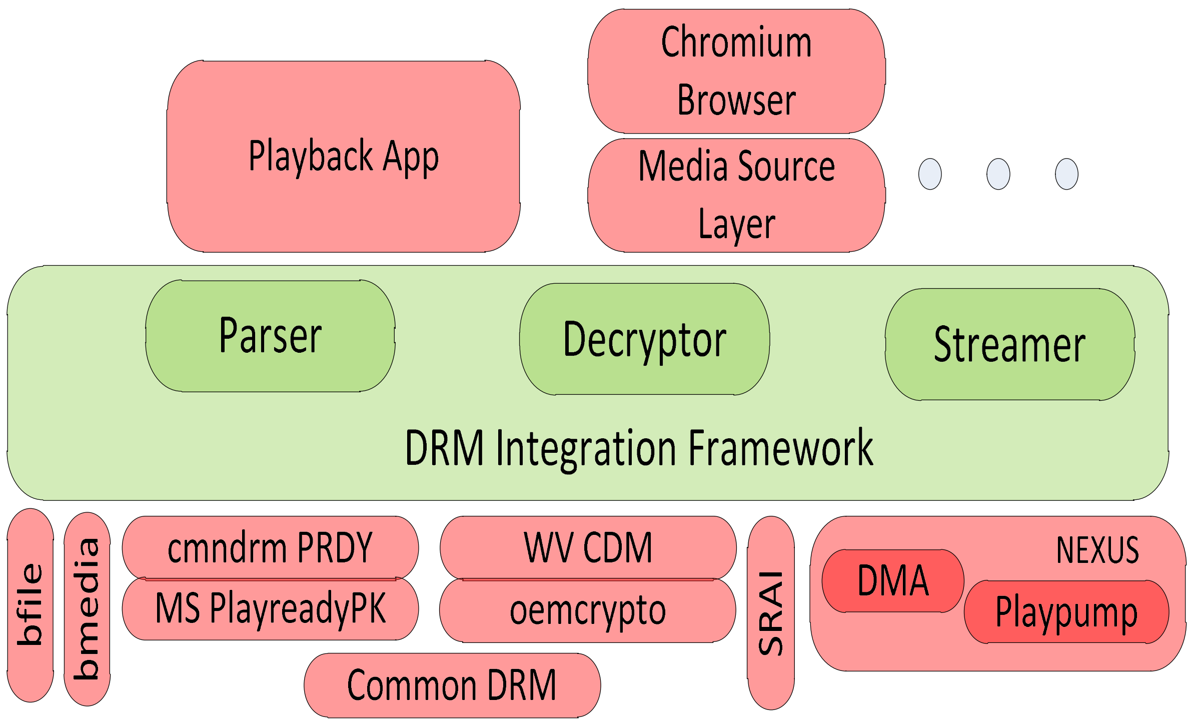
The goal to have this new DRM Integration Framework is to pack the common steps and routines into a separate module and provide application developers with easy-to-use API's to simplify their application development.

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.

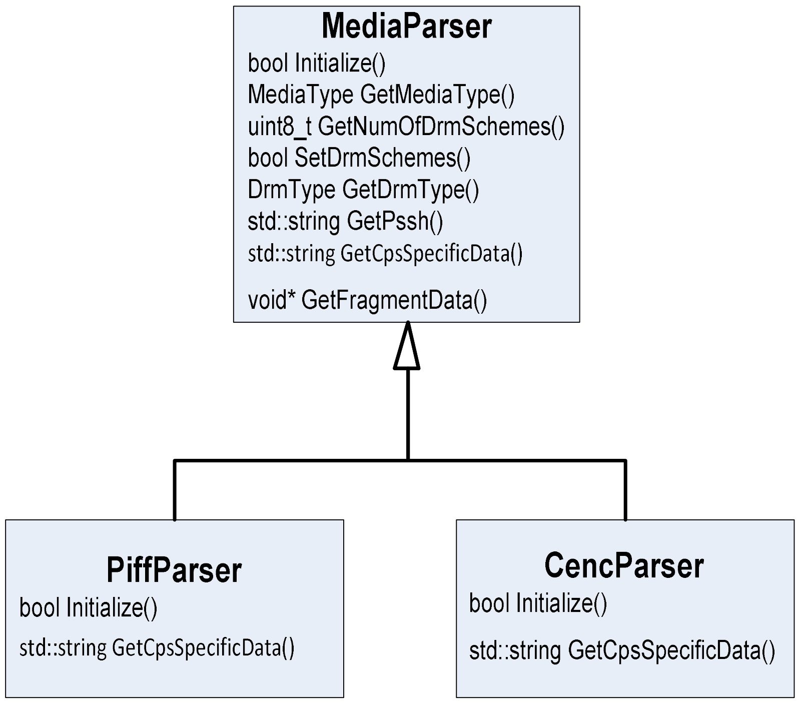
This diagram shows the position among related modules.



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

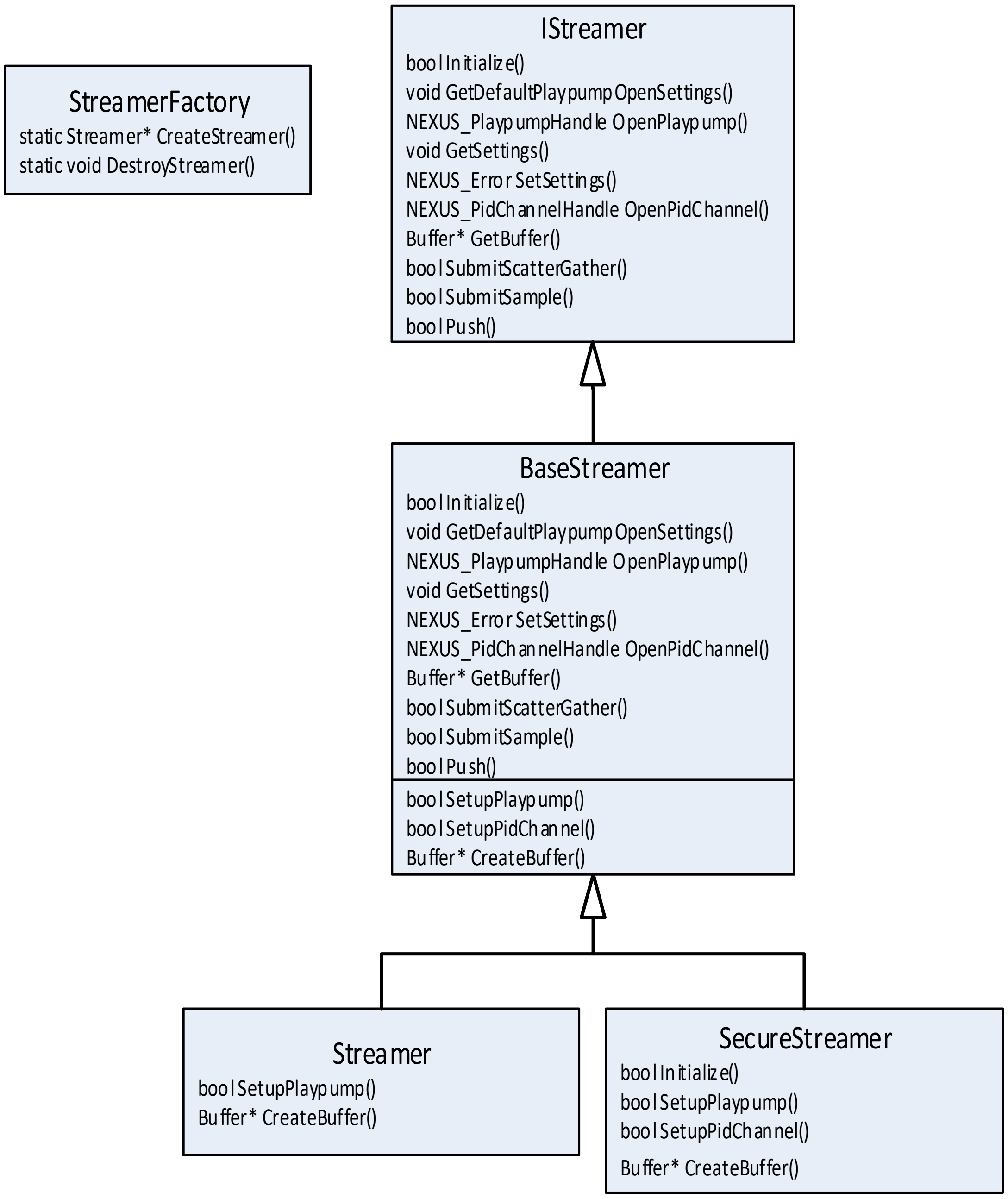
Streamer is used to push data to audio/video pipeline. Users can create either a SecureStreamer for SVP or a Streamer for non-SVP through StreamerFactory interface. Users can open one Nexus playpump using a Streamer and can’t open more than one playpump per Streamer.

Users can get the default NEXUS_PlaypumpOpenSettings through Streamer’s GetDefaultPlaypumpOpenSettings() , modify the settings and give it back to Streamer through OpenPlaypump().

Similarly users can get the current settings of NEXUS_PlaypumpSettings through GetSettings(). If any of the settings are modified, users have to set it back through Streamer’s SetSettings().

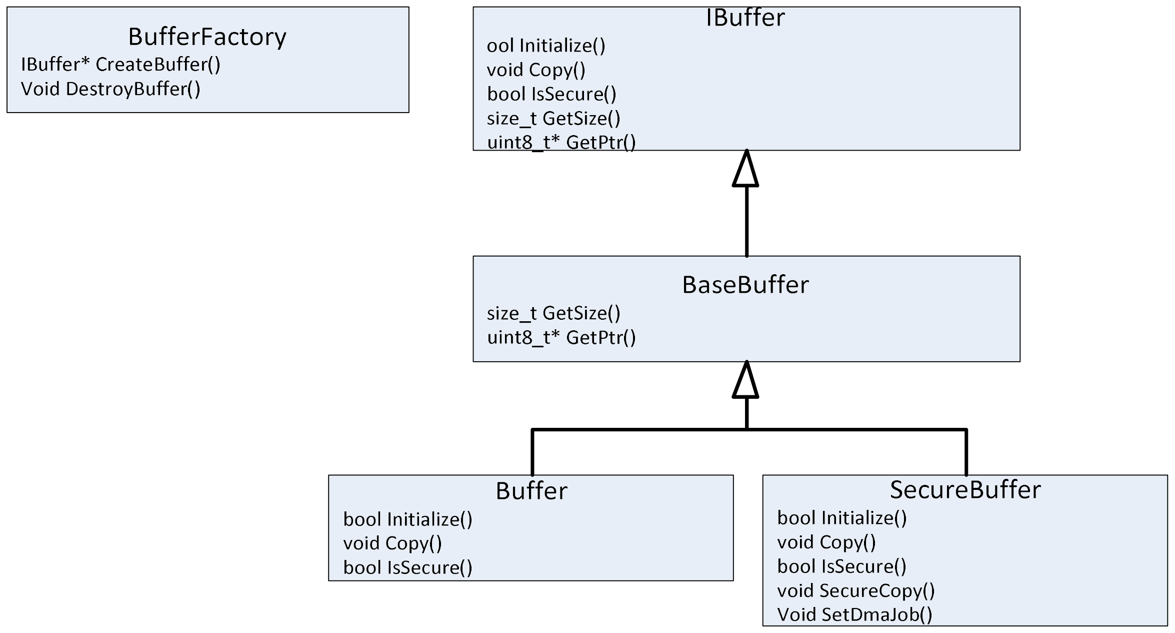
Scatter-gather feature was added in URSR 18.2 and later. Users can hold multiple data in different memory addresses/buffers and call the new SubmitScatterGather() API for each data. The underlying modules will pick up those data, so users don’t need to copy/transfer data to pack them into one contiguous memory area.

For encrypted media, users can call SubmitSample() API after Decryptor's DecryptSample() is done.



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.

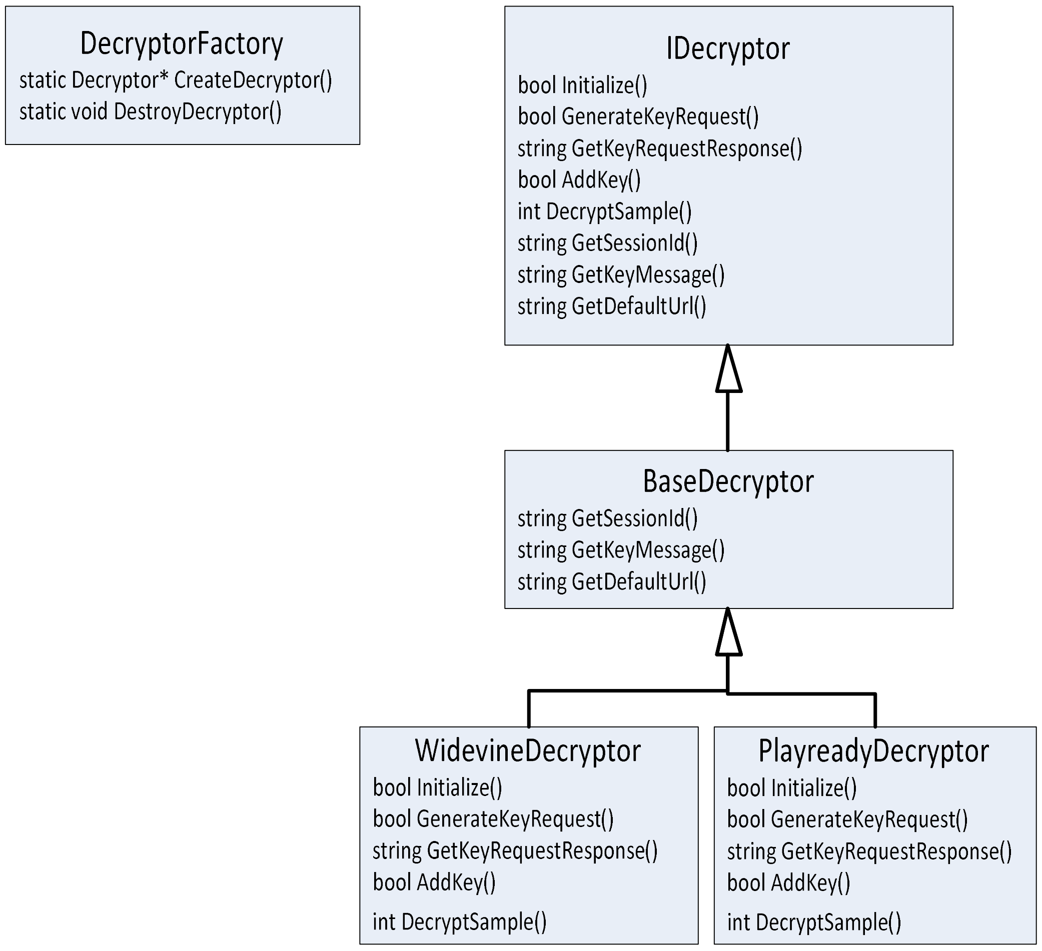


5.4 Decryptor

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.



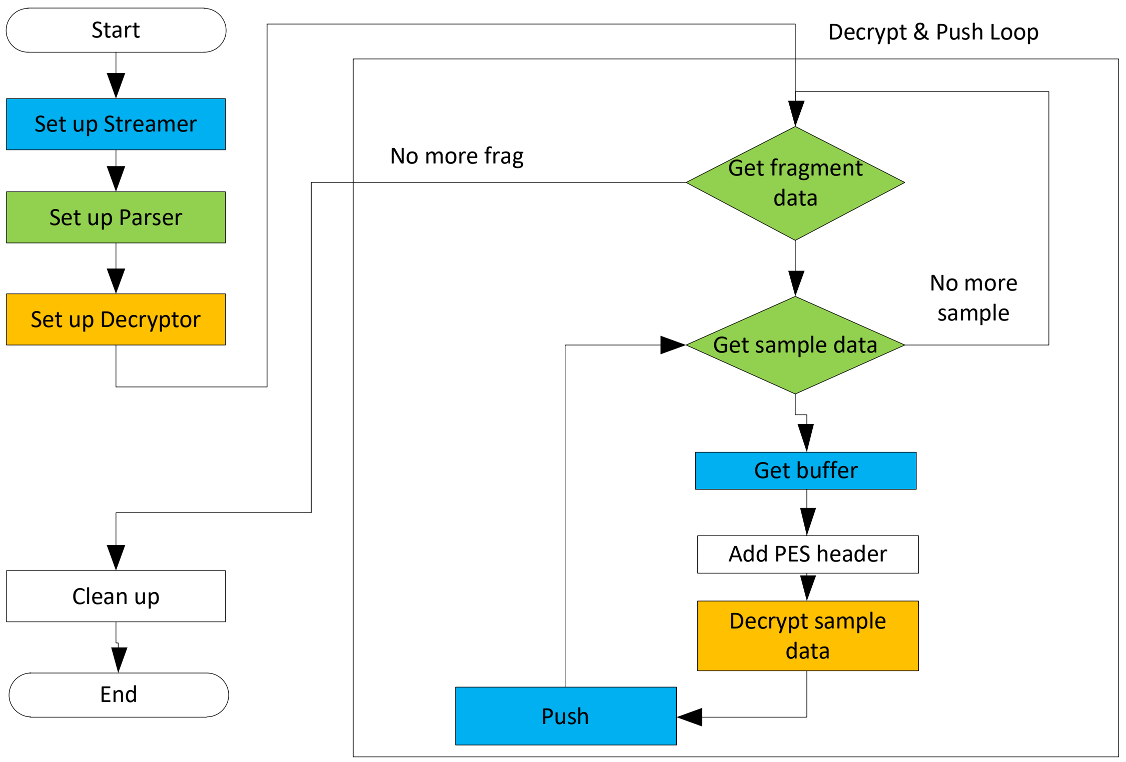
6 Application Development Guideline

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 retrieve PlaypumpOpenSettings through Streamer's API, configure it with own settings, and then give it back to Streamer through OpenPlaypump(). Similarly the application can get the current settings of NEXUS_PlaypumpSettings through GetSettings(). If any of the settings are modified, the application has to set it back through Streamer's SetSettings(). 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);

    // get current playpump settings, modify and set it back to the Streamer
    videoStreamer->GetSettings(&playpumpSettings);
    playpumpSettings.dataCallback.callback = play_callback;
    playpumpSettings.dataCallback.context = s_app.event;
    playpumpSettings.transportType = NEXUS_TransportType_eMpeg2Pes;
    videoStreamer->SetSettings(&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

As mentioned earlier, scatter-gather feature was added to Streamer in URSR 18.2 and later while the traditional copy way is still available. However it is not a good idea to mix both ways with the same Streamer. Below describes both ways.

6.5.1 Get fragment information

CENC/PIFF files include moof boxes together with mdat boxes which represent fragments. Each fragment includes multiple samples which represent frames. The application can get fragment information through Parser's GetFragmentData() and go on to process each sample data as shown in the below example. The below frag_info includes information regarding the fragment and each sample. Each sample's information includes sample size, the number of sub-samples, size of clear/encrypted data, etc. The below decoder_data returned from GetFragmentData() includes codec information and the application can use it to create audio/video codec headers utilizing bmedia functions.

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

        // process each sample in the fragment
        for (unsigned i = 0; i < frag_info->samples_info->sample_count; i++) {
...
            pSample = &frag_info->samples_info->samples[i];
            sampleSize = frag_info->sample_info[i].size;

            switch(frag_info->trackType)
            {
                // process video sample
                case BMP4_SAMPLE_ENCRYPTED_VIDEO:
                case BMP4_SAMPLE_AVC:
                {
...
                // decoder_data is used to create AV codec headers
                bmedia_read_h264_meta(&meta, decoder_data, decoder_len);
                bmedia_copy_h264_meta_with_nal_vec(dst, &meta, &bmp4_nal_vec);
                batom_cursor_from_accum(&cursor, dst);
                *hdr_len = batom_cursor_copy(&cursor, avcc_hdr, BMP4_MAX_PPS_SPS);
...
```

6.5.2 Handle PES header

This step has to be done at the beginning of handling each sample data.

Traditional copy way

The application has 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;
```

Scatter-gather way

The application has to submit the PES header through Streamer's SubmitScatterGather() API.

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

    // submit PES header to Streamer
    streamer->SubmitScatterGather(app->pVideoHeaderBuf, pes_header_len);

    bytes_processed += pes_header_len;
```

No copies or buffer operations are required with scatter-gather. However, the application has to keep the PES header data from being overwritten because the underlying module will directly look at this data. To avoid the buffer being overwritten, we recommend the application set up enough size of circular buffers with the number of entries greater than half of playpump's descFifoSize (default=100).

6.5.3 Clear data

If the sample data is clear, this step is very similar to the PES header handling.

Traditional copy way

Just copy the clear data to the Buffer that is gotten through Streamer's GetBuffer().

```
    // 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;
```

Scatter-gather way

Call Streamer's SubmitScatterGather() with the clear data. The original clear data must be kept from being overwritten until underlying playpump processes it. So the Buffer instance (input) below should not be destroyed right away. We recommend the application set up enough size of circular buffers with the number of entries greater than half of playpump's descFifoSize (default=100).

```
    // create a Buffer with the clear sample data
    IBuffer* input = BufferFactory::CreateBuffer(sampleSize, sampleData);

    // call SubmitScatterGather with the buffer
    // the second parameter "true" means this is the last data in the sample
    streamer->SubmitScatterGather(input, true);

    bytes_processed += sampleSize;
```

6.5.4 Encrypted data

Traditional copy way

If the sample data is encrypted, the application has to prepare two Buffers – one for input encrypted data and the other for output decrypted data. Decryptor's DecryptSample() 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);

    // copy all data from input into decOutput
    // this is required if input is partially clear and partially encrypted
    decOutput->Copy(0, input, 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;
```

Scatter-gather way

The application still has to prepare two Buffers, however, the output buffer is not gotten through Streamer's GetBuffer(). It just needs to be allocated from the same region as the Streamer (note the third parameter for the CreateBuffer() call). After DecryptSample() is done, data is decrypted into decOutput. However, input may include clear data partially. SubmitSample() will take care of both clear data and decrypted data from input and decOutput.

Similarly both input and decOutput must be kept from being overwritten until underlying playpump processes it. They should not be destroyed right away. We recommend the application set up enough size of circular buffers for both input and output with the number of entries greater than half of playpump's descFifoSize (default=100).

```
    // set up input buffer for decrypt (encrypted data)
    IBuffer* input = BufferFactory::CreateBuffer(sampleSize, sampleData);
...
    // set up output buffer in the same region as the Streamer
    IBuffer* decOutput = BufferFactory::CreateBuffer(sampleSize, NULL, streamer-
>IsSecure());

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

    // call Streamer's SubmitSample
    streamer->SubmitSample(pSample, input, decOutput);

    bytes_processed += numOfByteDecrypted;
```

6.5.5 Push data

After one sample data is processed as described above, the application can just call Streamer's 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;
```