



Freescale Parser Library API Specification

Version: 1.0

Date: July. 21, 2010

Freescale Semiconductor, Shanghai Software Development Center
UnitA, 20F, 560#, SongTao Road, Pu Dong New District
Shanghai, 201203, P.R of China

Revision History

Version	Date	Revised By	Description of Changes
1.0	07/21/2010	Amanda Lin	Initial version.

1. Introduction.....	5
1.1. In this document.....	5
1.2. References.....	5
2. API Functions	5
2.1. Query a Parser's Interface.....	5
2.2. Creation and Deletion	6
2.2.1. Get parser Version	6
2.2.2. Create Parser	6
2.2.3. Delete Parser	7
2.3. Index Loading	7
2.3.1. Initialize Index	7
2.3.2. Export Index.....	7
2.3.3. Import Index.....	8
2.4. Movie Properties.....	8
2.4.1. Movie Duration.....	8
2.4.1. Support Seeking or not.....	9
2.4.1. User Data	9
2.4.2. Number of Tracks	10
2.4.1. Number of Programs.....	10
2.4.1. Number of Tracks in One Program.....	10
2.5. Track Properties	11
2.5.1. Track Type.....	11
2.5.2. Track Duration	11
2.5.1. Language.....	12
2.5.1. Average Bit Rate.....	12
2.5.2. Decoder Specific Information.....	13
2.5.3. Video Properties.....	13
2.5.4. Audio Properties	14
2.5.5. Text (Subtitle) Properties.....	16
2.6. Reading Settings	17
2.6.1. Get Reading Mode	18
2.6.1. Set Reading Mode.....	18
2.6.2. Enable a track.....	19
2.7. Normal Playback.....	19
2.7.1. Track-based sample reading	19
2.7.2. File-based sample reading.....	21
2.8. Seek and Trick Mode	22
2.8.1. Seek.....	22
2.8.2. Track-based Trick Mode.....	23
2.8.3. File-based Trick Mode.....	25
3. Data Types & Constants	26
3.1. Error Codes	26
3.2. API Function ID List	27
3.3. Handle of Core Parser.....	28
3.4. Other Constants.....	28
3.4.1. Media Types.....	29

3.4.2. User data ID	29
4. Callback Functions.....	29
4.1. File I/O callback.....	29
4.2. Memory operation callback	31
4.3. Output buffer Request Callback.....	31
5. API Calling Sequence	32

1. Introduction

1.1. In this document

This document is the common API specification for the following core parser libraries:

- ASF
- AVI
- FLV
- MKV
- MP4
- MPEG-2 PS & TS
- OGG
- Real

The calling sequence of the API functions is also explained.

The API is declared in the header file “fsl_parser.h”.

1.2. References

Specification of supported media file formats.

* To get full DRM features (eg. DivX DRM, WMDRM), the right packages are needed to work with core parsers.

2. API Functions

2.1. Query a Parser's Interface

```
int32 FslParserQueryInterface(uint32 id, void ** func);
```

Description:

A core parser library shall be built as a shared library and this function must be implemented as the “entry function”

It's the 1st function to call after the parser shared library is opened. And it can return all other API function pointers implemented by the core parser, such as creating/deleting the core parser or reading media samples.

All other API functions shall obey the prototypes defined in the left part of this section.

Arguments:

id [in] The ID of the API function to query.

func [out] The related API function pointer. It must obey the prototype of the function ID. If the related API is optional and not implemented by the core parser, set this value to NULL.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.2. Creation and Deletion

All functions in this section must be implemented.

2.2.1. Get parser Version

```
typedef const char * (*FslParserVersionInfo)();
```

Description:

Function to get the core parser version.

API ID: PARSER_API_GET_VERSION_INFO

Return value:

ASCII Version string like "xxx_PARSER_xx.xx.xx".

2.2.2. Create Parser

```
typedef int32 (*FslCreateParser)( bool isLive,  
                                FslFileStream * streamOps,  
                                ParserMemoryOps * memOps,  
                                ParserOutputBufferOps * outputBufferOps,  
                                void * context,  
                                FslParserHandle * parserHandle);
```

Description:

Function to create the core parser. If the media file is supported by the core parser and not corrupted, this function will succeed.

API ID: PARSER_API_CREATE_PARSER

Arguments:

isLive [in] TRUE for a live source and FALSE for a local file source.

streamOps [in] Callback functions table for file I/O.
It implements the functions to open, close, seek and read the source file.

memOps [in] Memory operation callback table.
It implements the functions to malloc, calloc, realloc and free memory.

outputBufferOps [in] Output callback function table.
It implements the functions to request and release an output buffer.

context [in] Application context for the callback functions. It will be used as an argument of the callback functions. The core parser will never modify its content.

parserHandle [out] Handle of the core parser if the core parser is created successfully.
NULL for failure.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.2.3. Delete Parser

```
typedef int32 (*FslDeleteParser)(FslParserHandle parserHandle);
```

Description:

function to delete the core parser. This is the last API to call. API ID:

PARSER_API_DELETE_PARSER

Arguments:

parserHandle [in] Handle of the core parser.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.3. Index Loading

If the core parser can export and import the index table, it shall implement all functions in this section. The application can let parser load index table from the media file and export the index table in auto scan. When playing the media file, the application can import the index table very quickly thus speed up the file loading.

2.3.1. Initialize Index

```
typedef int32 (*FslParserInitializeIndex)(FslParserHandle parserHandle);
```

Description:

Function to initialize the index table by loading the index from the media file. For a movie played for the 1st time, the index table has to be loaded from the file. The index table increases with the movie duration. So the longer the movie is, the more time it takes to load the index.

Seeking and trick mode can be performed on a movie only if it has an index. But normal playback may not depend on the index. So even if this function fails, normal playback can still work as long as movie data are right for some file formats.

API ID: PARSER_API_INITIALIZE_INDEX

Arguments:

parserHandle [in] Handle of the core parser.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.3.2. Export Index

```
typedef int32 (*FslParserExportIndex)( FslParserHandle parserHandle,
```

```
uint8 * buffer,  
uint32 *size);
```

Description:

Function to export the index table of movie, after the index is loaded from the media file.
The application shall at first query the buffer size needed and then export the index data.

API ID: PARSER_API_EXPORT_INDEX

Arguments:

parserHandle [in] Handle of the core parser.

buffer [in] Buffer to export the index data. If this parameter is NULL, the core parser will return the size of buffer needed without exporting the index data.

size [in/out] Size of the buffer as input, in bytes.
It shall not be smaller than the needed size.

Size of the index data in the buffer as output, in bytes.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.3.3. Import Index

```
typedef int32 (*FslParserImportIndex)( FslParserHandle parserHandle,  
uint8 * buffer,  
uint32 size);
```

Description:

Function to import the index table of a track from outside, instead of loading it from the file.
This can reduce file loading time if the movie index table has been exported before in auto-scan phase..

API ID: PARSER_API_IMPORT_INDEX

Arguments:

parserHandle [in] Handle of the core parser.

buffer [in] Buffer containing the index data.

size [in] Size of the index data in the buffer, in bytes.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.4. Movie Properties

2.4.1. Movie Duration

```
typedef int32 (*FslParserGetMovieDuration)(FslParserHandle parserHandle, uint64 * usDuration);
```

Description:

Function to tell the movie duration. Usually it's equal to the longest track's duration if multiple tracks are present.

Arguments:

parserHandle [in]	Handle of the core parser.
usDuration [out]	Duration in us. If the movie duration is unknown, set this value to PARSE_UNKNOWN_DURATION (0).

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.4.1. Support Seeking or not

```
typedef int32 (*FslParserIsSeekable)(FslParserHandle parserHandle, bool * seekable);
```

Description:

Function to tell whether the movie can support seeking and trick mode (FF/RW). Usually if a movie has index table, it can support seeking and trick mode. Otherwise it can only support sequential playback .

Arguments:

parserHandle [in]	Handle of the core parser.
seekable [out]	TRUE if seeking and trick mode are supported and FALSE if not.

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.4.1. User Data

```
typedef int32 (*FslParserGetUserData)( FslParserHandle parserHandle,  
                                     uint32 userDataId,  
                                     uint16 ** unicodeString,  
                                     uint32 * stringLength);
```

Description:

Function to tell the user data information (title, artist, genre etc) of the movie.

Arguments:

parserHandle [in]	Handle of the core parser.
-------------------	----------------------------

id [in] Type of the user data.

USER_DATA_TITLE, USER_DATA_GENRE, USER_DATA_ARTIST ...

unicodeString [out]	Buffer containing the information, a null-terminated Unicode string. The core parser manages this buffer and the user shall NOT free it. If no such info is available, this value shall be set to NULL.
---------------------	---

stringLength [out]	Length of the string , which is the number of Unicode characters, not including the terminator. This value is 0 if the string is empty.
--------------------	--

Return value:

PARSER_SUCCESS - Success

Other error codes – Failure

2.4.2. Number of Tracks

```
typedef int32 (*FslParserGetNumTracks)(FslParserHandle parserHandle, uint32 * numTracks);
```

Description:

Function to tell how many tracks there are in the movie. If a media file format does not support multiple programs, the core parser must implement this API.

Arguments:

parserHandle [in] Handle of the core parser.

numTracks [out] Number of tracks.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.4.1. Number of Programs

```
typedef int32 (*FslParserGetNumPrograms)( FslParserHandle parserHandle,
                                           uint32 * numPrograms);
```

Description:

Function to tell how many programs there are in the movie. If a media file format support multiple programs such as MPEG-2 Transport Stream, the core parser must implement this API.

Arguments:

parserHandle [in] Handle of the core parser.

numPrograms [out] Number of programs.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.4.1. Number of Tracks in One Program

```
typedef int32 (*FslParserGetProgramTracks)( FslParserHandle parserHandle,
                                           uint32 programNum,
                                           uint32 * numTracks,
                                           uint32 * pTrackNums);
```

Description:

Function to tell how many tracks there are in one program. If a media file format supports multiple programs, the core parser must implement this API.

Arguments:

parserHandle [in] Handle of the core parser.

programNum [in] Number of the program to check, 0-based.

numTracks [out] Number of tracks in the specified program.

pTrackNums [out] Track number list in the specified program.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.5. Track Properties

2.5.1. Track Type

```
typedef int32 (*FslParserGetTrackType)( FslParserHandle parserHandle,
                                         uint32 trackNum,
                                         uint32 * mediaType,
                                         uint32 * decoderType,
                                         uint32 * decoderSubtype);
```

Description:

Function to tell the type of a track, includes the media type, decoder type and decoder subtype if available.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based.

mediaType [out] Media type of the track. (video, audio, subtitle...)
"MEDIA_TYPE_UNKNOWN" means the media type is unknown.

decoderType [out] Decoder type of the track if available. (eg. MPEG-4, H264, AAC, MP3, AMR ...)
"UNKNOWN_CODEC_TYPE" means the decoder type is unknown.

decoderSubtype [out] Decoder Subtype type of the track if available. (eg. AMR-NB, AMR-WB ...)
"UNKNOWN_CODEC_SUBTYPE" means the decoder subtype is unknown.

Return value:

PARSER_SUCCESS - Success

Other error codes – Failure

2.5.2. Track Duration

```
typedef int32 (*FslParserGetTrackDuration)( FslParserHandle parserHandle,
                                             uint32 trackNum,
                                             uint64 * usDuration);
```

Description:

Function to tell a track's duration.

The tracks may have different durations. And the movie's duration is usually the video track's duration (maybe not the longest one).

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based.

usDuration [out] Duration in us. If the track duration is unknown, set this value to
PARSER_UNKNOWN_DURATION (0).

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.5.1. Language

```
typedef int32 (*FslParserGetLanguage)( FslParserHandle parserHandle,  
                                       uint32 trackNum,  
                                       uint8 * threeCharCode);
```

Description:

Function to tell the language of a track used.
This is helpful to select an audio/subtitle track or menu pages.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based.

threeCharCode [out] Three or two character language code.

See ISO 639-2/T for the set of three character codes. Eg. 'eng' for English.

Four special case:
mis- "uncoded languages"
mul- "multiple languages"
und- "undetermined language"
zxx- "no linguistic content"

See ISO 639 for the set of two character codes. Eg. 'en' for English and 'zh' for Chinese.

If ISO 639 is used, the 3rd character shall be '\0'.
If ISO 639-T is used, the 4th character shall be '\0'.

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.5.1. Average Bit Rate

```
typedef int32 (*FslParserGetBitRate)( FslParserHandle parserHandle,  
                                       uint32 trackNum,  
                                       uint32 * bitrate);
```

Description:

Function to tell the average bit rate of a track.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based.

bitrate [out] Bit rate of the track. For CBR stream, this is the real bit rate.
If the average bit rate is unknown, set this value to ZERO.

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.5.2. Decoder Specific Information

```
typedef int32 (*FslParserGetDecSpecificInfo)( FslParserHandle parserHandle,  
                                              uint32 trackNum,  
                                              uint8 ** data,  
                                              uint32 * size);
```

Description:

Function to tell the decoder specific information of a track.
The content of the information is type dependent and defined in the encoding level.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based.

data [out] Buffer holding the decoder specific information.
The user shall never free this buffer.

size [out] Size of the decoder specific information, in bytes.

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.5.3. Video Properties

2.5.3.1. Frame Width

```
typedef int32 (*FslParserGetVideoFrameWidth)(FslParserHandle parserHandle,  
                                              uint32 trackNum,  
                                              uint32 *width);
```

Description:

Function to tell the frame width in pixels of a video track.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be a video track.

width [out] Frame width in pixels.

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.5.3.2. Frame Height

```
typedef int32 (*FslParserGetVideoFrameHeight)(FslParserHandle parserHandle,  
                                              uint32 trackNum,
```

```
uint32 *height);
```

Description:

Function to tell the frame height in pixels of a video track.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be a video track.

height [out] Frame height in pixels.

Return value:

PARSER_SUCCESS - Success

Other error codes – Failure

2.5.3.3. Frame Rate

```
typedef int32 (*FslParserGetVideoFrameRate)( FslParserHandle parserHandle,  
                                              uint32 trackNum,  
                                              uint32 * rate,  
                                              uint32 * scale);
```

Description:

Function to tell the frame rate a video track, in the form of (rate / scale).

Arguments:

parserHandle [in] Handle of the core parser.

rate [out] Numerator of the frame rate. If the value is unknown, set it to ZERO.

scale [out] Denominator of the frame rate. If the value is unknown, set it to ZERO.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.5.4. Audio Properties

2.5.4.1. Number of Channels

```
typedef int32 (*FslParserGetAudioNumChannels)( FslParserHandle parserHandle,  
                                              uint32 trackNum,  
                                              uint32 * numchannels);
```

Description:

Function to tell how many channels in an audio track.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be an audio track.

numchannels [out] Number of the channels. 1 mono, 2 stereo, or more for multiple channels.

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.5.4.2. Audio Sampling Rate

```
typedef int32 (*FslParserGetAudioSampleRate)( FslParserHandle parserHandle,  
                                              uint32 trackNum,  
                                              uint32 * sampleRate);
```

Description:

Function to tell the audio sample rate (sampling frequency) of an audio track.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be an audio track.

sampleRate [out] Audio integer sample rate (sampling frequency).

Return value:

PARSER_SUCCESS - Success

Other error codes - Failure

2.5.4.3. Bit Depth

```
typedef int32 (*FslParserGetAudioBitsPerSample)(FslParserHandle parserHandle,  
                                              uint32 trackNum,  
                                              uint32 * bitsPerSample);
```

Description:

Function to tell the bits per sample (bit depth) for a PCM audio sample.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be an audio track.

bitsPerSample [out] Bits per PCM sample. If this value is unknown, set it to ZERO.

Return value:

PARSER_SUCCESS - Success

Other error codes – Failure

2.5.4.4. Block Alignment

```
typedef int32 (*FslParserGetAudioBlockAlign)( FslParserHandle parserHandle,  
                                              uint32 trackNum,  
                                              uint32 * blockAlign);
```

Description:

Function to tell the block alignment for an audio track. It's an optional API.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be an audio track.

blockAlign [out] Block alignment in bytes.

Return value:

PARSER_SUCCESS - Success

Other error codes – Failure

2.5.4.5. Bits per Frame

```
typedef int32 (*FslParserGetAudioBitsPerFrame)( FslParserHandle parserHandle,
                                                uint32 trackNum,
                                                uint32 *bits_per_frame);
```

Description:

Function to tell the bits per frame. It's an optional API and only for Real Audio.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be an audio track.

bits_per_frame [out] Bits per frame.

Return value:

PARSER_SUCCESS - Success

Other error codes – Failure

2.5.4.6. Channel Mask

```
typedef int32 (*FslParserGetAudioChannelMask)( FslParserHandle parserHandle,
                                                uint32 trackNum,
                                                uint32 * channelMask);
```

Description:

Function to tell the channel mask. It's an optional API and only for WMA Audio.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be an audio track.

channelMask [out] Channel mask.

Return value:

PARSER_SUCCESS - Success

Other error codes – Failure

2.5.5. Text (Subtitle) Properties

2.5.5.1. Text Window Width

```
typedef int32 (*FslParserGetTextTrackWidth)( FslParserHandle parserHandle,
                                                uint32 trackNum,
                                                uint32 * width);
```

Description:

Function to tell the width of a text (subtitle) track.

The text track defines a window to display the subtitles.
This window shall be positioned in the middle of the screen.
And the sample is displayed in the window. How to position the sample within the window is defined by the sample data.
T

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be a text track.

width [out] Width of the text track, in pixels.

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.5.5.2. Text Window Height

```
typedef int32 (*FslParserGetTextTrackHeight)( FslParserHandle parserHandle,  
                                              uint32 trackNum,  
                                              uint32 * height);
```

Description:

Function to tell the height of a text (subtitle) track.
The text track defines a window to display the subtitles.
This window shall be positioned in the middle of the screen.
And the sample is displayed in the window. How to position the sample within the window is defined by the sample data.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track, 0-based. It must be a text track.

height [out] Height of the text track, in pixels.

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.6. Reading Settings

A core parser may support two reading modes: file-based or track-based reading mode

File-based reading mode:

This mode is usually more efficient, especially when the file system cache is not so big. Different track's data output are interleaved in the same order as their layout in the file, The core parser only reads the file once and avoids unwanted file seeking.
But this mode cannot handle non-interleaved or deep-interleaved media file because the decoding queue depth restrictions on most frameworks. The application need to handle this risk.
And it's recommended the core parser not support file-based reading mode for non- or deep-interleaved files.

For a live source, file-based reading mode is preferred because the time used on file seeking is not acceptable.

Track-based reading mode:

Under this mode, the core parser will output samples of a specified track in its own time order. There is no risk of decoder queue overflow for deep-interleaved files, and no risk of pre-roll failure after seeking near EOF. But it's usually less efficient than file-based reading mode when multiple tracks are present and occasional file-seeking is unavoidable.

After the core parser is created and index table loaded, the core parser will have a default reading mode. And the application can try to change the reading mode as long as the new mode is supported by the core parser.

2.6.1. Get Reading Mode

```
typedef int32 (*FslParserGetReadMode)(FslParserHandle parserHandle, uint32 * readMode);
```

Description:

Function to tell the current reading mode.

A core parser may support both file-based or track-based reading mode, or only one reading mode. After the core parser is created and index table loaded, the core parser will have a default reading mode.

Arguments:

parserHandle [in]	Handle of the core parser.
readMode [out]	Current reading mode. PARSER_READ_MODE_FILE_BASED or PARSER_READ_MODE_TRACK_BASED

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.6.1. Set Reading Mode

```
typedef int32 (*FslParserSetReadMode)(FslParserHandle parserHandle, uint32 readMode);
```

Description:

Function to set a reading mode.

A core parser may support both file-based or track-based reading mode, or only one reading mode. After the parser is created and index table loaded, the core parser will have a default reading mode. The application can try to change the reading mode as long as the new mode is supported by the core parser.

If the mode is not supported by the parser, this function will fail and core parser still works under the original mode.

Arguments:

parserHandle [in]	Handle of the core parser.
readMode [in]	Current reading mode. PARSER_READ_MODE_FILE_BASED or PARSER_READ_MODE_TRACK_BASED

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.6.2. Enable a track

```
typedef int32 (*FslParserEnableTrack)(FslParserHandle parserHandle,  
                                     uint32 trackNum,  
                                     bool enable);
```

Description:

Function to enable or disable a track.

The parser can only output samples from enabled tracks.

- Under file-based reading mode, all disabled tracks will be skipped automatically by the core parser.
- Under track-based reading mode, reading a disabled track will fail with err PARSER_ERR_TRACK_DISABLED.

To avoid unexpected memory cost or data output from a track, the application can disable unwanted tracks.

Arguments:

parserHandle [in]	Handle of the core parser.
trackNum [in]	Number of the track, 0-based.
enable [in]	TRUE to enable a track and FALSE to disable a track.

Return value:

PARSER_SUCCESS - Success
Other error codes – Failure

2.7. Normal Playback

To support normal playback (rate = 1X), parser provides two API for file-based and track-based reading mode respectively.

Core parser supports partial output of large samples, piece by piece. The exception is some DRM-encrypted clips that need a big enough output buffer for data decryption.

2.7.1. Track-based sample reading

```
typedef int32 (*FslParserGetNextSample)(FslParserHandle parserHandle,  
                                       uint32 trackNum,  
                                       uint8 ** sampleBuffer,  
                                       void ** bufferContext,  
                                       uint32 * dataSize,  
                                       uint64 * usStartTime,  
                                       uint64 * usDuration,  
                                       uint32 * sampleFlags);
```

Description:

Function to read the next sample from a specified track. If the parser support track-based reading mode, it shall implement this API.

The data reading is track-based. Given the track number, the parser can output any enabled track's samples one by one. It makes the switching easy among multiple audio/subtitles.

It supports partial output of large samples. The core parser will use call back to request an output buffer from the application. If the output is not big enough to put the entire sample, the parser can output a piece of data and the remaining data can be got by repetitive calling this function until the whole sample is output.

Arguments:

parserHandle [in]	Handle of the core parser.
trackNum [in]	Number of the track to read, 0-based.
sampleBuffer [out]	Output buffer holding the sample data. It's the buffer got in one previous "request buffer" callback function.
bufferContext [out]	Buffer context, got in one previous "request buffer" callback function. It's usually an application-level buffer structure and the core parser never change it.
dataSize [out]	If an entire sample or part of a sample is output successfully (return value is <code>PARSER_SUCCESS</code>), it's the size of the data actually got, in bytes.
usStartTime [out]	Start time of the sample in us (timestamp). If this value is unknown, set it to <code>PARSER_UNKNOWN_TIME_STAMP</code> (-1).
usDuration [out]	Sample duration in us. If this value is unknown, set it to <code>PARSER_UNKNOWN_DURATION</code> (0).
sampleFlags [out]	Flags of this sample if a sample is got successfully. <code>FLAG_SYNC_SAMPLE</code> Whether this sample is a sync sample (key frame). For non-video media, the core parser shall take every sample as sync sample. <code>FLAG_SAMPLE_ERR_CONCEALED</code> There is error in bitstream but a sample is still got by error concealment. <code>FLAG_SAMPLE_SUGGEST_SEEK</code> A seeking on ALL tracks is suggested although samples can be got by error concealment. Because there are many corrupts, and A/V sync is likely impacted by error concealment(eg. scanning bitstream). <code>FLAG_SAMPLE_NOT_FINISHED</code> Sample data output is not finished because the output buffer is not big enough. Need to get the remaining data by repetitive calling this function.

Return value:

`PARSER_SUCCESS` - An entire sample or part of it is got successfully.

`PARSER_EOS` - No sample is got because of end of the track.

PARSER_INSUFFICIENT_MEMORY - Buffer is too small to hold the sample.
For very special cases when partial output is not applicable.

PARSER_READ_ERROR - File reading error. No need for further error concealment.

PARSER_ERR_CONCEAL_FAIL - There is error in bitstream, and no sample can be found
by error concealment. A seeking is helpful.

Others - Other failures.

2.7.2. File-based sample reading

```
typedef int32 (*FslParserGetFileNextSample)( FslParserHandle parserHandle,  
                                             uint32 * trackNum,  
                                             uint8 ** sampleBuffer,  
                                             void ** bufferContext,  
                                             uint32 * dataSize,  
                                             uint64 * usStartTime,  
                                             uint64 * usDuration,  
                                             uint32 * sampleFlags);
```

Description:

Function to read the next sample from the movie. If the parser support file-based reading mode, it shall implement this API.

The data reading is file-based. The parser will output the next sample of the track in the current file position and tell the application which track is being output.

It supports partial output of large samples. The core parser will use call back to request an output buffer from the application. If the output is not big enough to put the entire sample, the parser can output a piece of data and the remaining data can be got by repetitive calling this function until the whole sample is output.

Arguments:

parserHandle [in]	Handle of the core parser.
trackNum [out]	Number of the track being read, 0-based.
sampleBuffer [out]	Output buffer holding the sample data. It's the buffer got in one previous "request buffer" callback function.
bufferContext [out]	Buffer context, got in one previous "request buffer" callback function. It's usually an application-level buffer structure and the core parser never change it.
dataSize [out]	If an entire sample or part of a sample is output successfully (return value is PARSER_SUCCESS), it's the size of the data actually got, in bytes.
usStartTime [out]	Start time of the sample in us (timestamp). If this value is unknown, set it to PARSER_UNKNOWN_TIME_STAMP (-1).
usDuration [out]	Sample duration in us. If this value is unknown, set it to PARSER_UNKNOWN_DURATION (0).

sampleFlags [out] Flags of this sample if a sample is got successfully.

FLAG_SYNC_SAMPLE

Whether this sample is a sync sample (key frame).

For non-video media, the core parser shall take every sample as sync sample.

FLAG_SAMPLE_ERR_CONCEALED

There is error in bitstream but a sample is still got by error concealment.

FLAG_SAMPLE_SUGGEST_SEEK

A seeking on ALL tracks is suggested although samples can be got by error concealment.

Because there are many corrupts, and A/V sync is likely impacted by error concealment(eg. scanning bitstream).

FLAG_SAMPLE_NOT_FINISHED

Sample data output is not finished because the output buffer is not big enough. Need to get the remaining data by repetitive calling this function.

Return value:

PARSER_SUCCESS - An entire sample or part of it is got successfully.

PARSER_EOS - No sample is got because of end of the movie.

PARSER_INSUFFICIENT_MEMORY - Buffer is too small to hold the sample.
For very special cases when partial output is not applicable.

PARSER_READ_ERROR - File reading error. No need for further error concealment.

PARSER_ERR_CONCEAL_FAIL - There is error in bit stream, and no sample can be found
by error concealment. A seeking is helpful.

Others - Other failures.

2.8. Seek and Trick Mode

Seeking and trick mode (FF/RW) can be performed only on a seekable movie.

If a movie is not seekable, the core parser only need support seeking to time 0 - start of movie.

2.8.1. Seek

```
typedef int32 (*FslParserSeek)( FslParserHandle parserHandle,  
                                uint32 trackNum,  
                                uint64 * usTime,  
                                uint32 flag);
```

Description:

Function to seek a track to a target time. It will seek to a sync sample of the time stamp matching the target time. Due to the scarcity of the video sync samples (key frames), there can be a gap between the target time and the timestamp of the matched sync sample. So this time stamp will be output to as the accurate start time of the following playback segment.

NOTE:

(1) Seeking to the beginning of the movie (target time is 0 us) does not require the index table.

- (2) For some media formats, seeking is always file-based and the control flag can not accurately control the matched time for non-primary tracks. A safe way for the application is to check the 1st time stamp after seeking.

Arguments:

parserHandle [in] Handle of the core parser.

trackNum [in] Number of the track to seek, 0-based.

usTime [in/out] Target time to seek as input, in us.
 Actual seeking time, timestamp of the matched sync sample, as output.

flag [in] Control flags to seek.

SEEK_FLAG_NEAREST

Default flag. The matched time stamp shall be the nearest one to the target time (may be later or earlier)

SEEK_FLAG_NO_LATER

The matched time stamp shall be no later than the target time.

SEEK_FLAG_NO_EARLIER

The matched time stamp shall be no earlier than the target time.

Return value:

PARSER_SUCCESS Seeking succeeds.

PARSER_EOS Seeking is outside the movie/track scope.

PARSER_ERR_NOT_SEEKABLE

Seeking fails because the movie is not seekable (index not available).

Others Seeking fails for other reason.

2.8.2. Track-based Trick Mode

```
typedef int32 (*FslParserGetNextSyncSample)( FslParserHandle parserHandle,
                                             uint32 direction,
                                             uint32 trackNum,
                                             uint8 ** sampleBuffer,
                                             void ** bufferContext,
                                             uint32 * dataSize,
                                             uint64 * usStartTime,
                                             uint64 * usDuration,
                                             uint32 * flags);
```

Description:

Function to get the next or previous sync sample (key frame) from a track. For trick mode FF/RW. If the parser support track-based reading mode and trick mode, it shall implement this API.

It supports partial output of large samples.

Arguments:

parserHandle [in] Handle of the core parser.

direction [in] Direction to get the sync sample.

FLAG_FORWARD

Read the next sync sample from current reading position.

FLAG_BACKWARD

Read the previous sync sample from current reading position.

trackNum [in] Number of the track to read, 0-based.

sampleBuffer [out] Output buffer holding the sample data.
It's the buffer got in one previous "request buffer" callback function.

bufferContext [out] Buffer context, got in one previous "request buffer" callback function.
It's usually an application-level buffer structure and the core parser never change it.

dataSize [out] If an entire sample or part of a sample is output successfully (return value is **PARSER_SUCCESS**), it's the size of the data actually got, in bytes.

usStartTime [out] Start time of the sample in us (timestamp). If this value is unknown, set it to **PARSER_UNKNOWN_TIME_STAMP** (-1).

usDuration [out] Sample duration in us. If this value is unknown, set it to **PARSER_UNKNOWN_DURATION** (0).

flag [out] Flags of this sample if a sample is got successfully.

FLAG_SYNC_SAMPLE

Whether this sample is a sync sample (key frame).

For non-video media, the wrapper shall take every sample as sync sample.

This flag shall always be SET for this API.

FLAG_SAMPLE_ERR_CONCEALED

There is error in bitstream but a sample is still got by error concealment.

FLAG_SAMPLE_SUGGEST_SEEK

A seeking on ALL tracks is suggested although samples can be got by error concealment.

Because there are many corrupts,

and A/V sync is likely impacted by simple concealment(scanning bitstream).

FLAG_SAMPLE_NOT_FINISHED

Sample data output is not finished because the buffer is not big enough. Need to get the remaining data by repetitive calling this function.

Return value:

PARSER_SUCCESS An entire sync sample or 1st part of it is got successfully.

PARSER_ERR_NOT_SEEKABLE No sync sample is got because the movie is not seekable (index not available).

PARSER_EOS Reaching the end of the track, no sync sample is got.

PARSER_BOS Reaching the beginning of the track, no sync sample is got.

PARSER_INSUFFICIENT_MEMORY Buffer is too small to hold the sample.

PARSER_READ_ERROR File reading error. No need for further error concealment.

PARSER_ERR_CONCEAL_FAIL There is error in bit stream, and no sample can be found by error concealment. A seeking is helpful.
Others ... Reading fails for other reason.

2.8.3. File-based Trick Mode

```
typedef int32 (*FslParserGetFileNextSyncSample)(FslParserHandle parserHandle,  
                                                uint32 direction,  
                                                uint32 * trackNum,  
                                                uint8 ** sampleBuffer,  
                                                void ** bufferContext,  
                                                uint32 * dataSize,  
                                                uint64 * usStartTime,  
                                                uint64 * usDuration,  
                                                uint32 * flags);
```

Description:

Function to get the next or previous sync sample (key frame) from the movie. If the parser support file-based reading mode and trick mode, it shall implement this API.

It supports partial output of large samples.

Arguments:

parserHandle [in] Handle of the core parser.

direction [in] Direction to get the sync sample.
 FLAG_FORWARD
 Read the next sync sample from current reading position.

 FLAG_BACKWARD
 Read the previous sync sample from current reading position.

trackNum [out] Number of the track being read, 0-based.

sampleBuffer [out] Output buffer holding the sample data.
 It's the buffer got in one previous "request buffer" callback function.

bufferContext [out] Buffer context, got in one previous "request buffer" callback function.
 It's usually an application-level buffer structure and the core parser never change it.

dataSize [out] If an entire sample or part of a sample is output successfully
 (return value is PARSER_SUCCESS), it's the size of the data
 actually got, in bytes.

usStartTime [out] Start time of the sample in us (timestamp). If this value is unknown,
 set it to PARSER_UNKNOWN_TIME_STAMP (-1).

usDuration [out] Sample duration in us. If this value is unknown, set it to
 PARSER_UNKNOWN_DURATION (0).

flag [out] Flags of this sample if a sample is got successfully.

FLAG_SYNC_SAMPLE

Whether this sample is a sync sample (key frame).

For non-video media, the wrapper shall take every sample as sync sample.

This flag shall always be SET for this API.

FLAG_SAMPLE_ERR_CONCEALED

There is error in bitstream but a sample is still got by error concealment.

FLAG_SAMPLE_SUGGEST_SEEK

A seeking on ALL tracks is suggested although samples can be got by error concealment.

Because there are many corrupts,

and A/V sync is likely impacted by simple concealment(scanning bitstream).

FLAG_SAMPLE_NOT_FINISHED

Sample data output is not finished because the buffer is not big enough. Need to get the remaining data by repetitive calling this function.

Return value:

PARSER_SUCCESS An entire sync sample or 1st part of it is got successfully.

PARSER_ERR_NOT_SEEKABLE No sync sample is got because the movie is not seekable (index not available).

PARSER_EOS Reaching the end of the track, no sync sample is got.

PARSER_BOS Reaching the beginning of the track, no sync sample is got.

PARSER_INSUFFICIENT_MEMORY Buffer is too small to hold the sample.

PARSER_READ_ERROR File reading error. No need for further error concealment.

PARSER_ERR_CONCEAL_FAIL There is error in bit stream, and no sample can be found by error concealment. A seeking is helpful.

Others ... Reading fails for other reason.

3. Data Types & Constants

3.1. Error Codes

Common Error Codes	Comments
PARSER_SUCCESS	success
PARSER_EOS	Reach end of the steam. Not an error.
PARSER_BOS	Reach beginning of the stream. Not an error.
PARSER_ERR_UNKNOWN	Unknown failure, not captured by parser logic
PARSER_NOT_IMPLEMENTED	The feature is not implemented yet.
PARSER_ERR_INVALID_PARAMETER	Invalid parameter.
PARSER_INSUFFICIENT_MEMORY	memory not enough, causing general memory allocation failure

PARSER_INSUFFICIENT_DATA	data not enough, parser need more data to go ahead
PARSER_ERR_NO_OUTPUT_BUFFER	can not get sample buffer for outputx loading.
PARSER_FILE_OPEN_ERROR	Can not open the movie file.
PARSER_READ_ERROR	Error on file reading. No need for further error concealment
PARSER_WRITE_ERROR	Error on file writing.
PARSER_SEEK_ERROR	Error on file seeking.
PARSER_ILLEGAL_FILE_SIZE	file size is wrong or exceeds parser's capacity.(some parser can not handle file larger than 2GB)
PARSER_ERR_INVALID_MEDIA	invalid or unsupported media format
PARSER_ERR_NOT_SEEKABLE	Perform seeking or trick mode on a movie NOT seekable.
PARSER_ERR_CONCEAL_FAIL	Error in bitstream and no sample can be found by error concealment. If the file is seekable, it's better to perform a seeking than further searching the bit stream for the next sample.
PARSER_ERR_TRACK_DISABLED	The track is disabled and no media samples can be read from it. Only enabled track can output samples.
PARSER_ERR_INVALID_READ_MODE	The reading mode is invalid, or some operation is illegal under current reading mode
DRM_ERR_NOT_AUTHORIZED_USER	Not an authorized user to play a DRM-protected file
DRM_ERR_RENTAL_EXPIRED	The protected rental file is expired (reaching its view limit)

3.2. API Function ID List

```
enum /* API function ID */
{
    /* creation & deletion */
    PARSER_API_GET_VERSION_INFO = 0,
    PARSER_API_CREATE_PARSER    = 1,
    PARSER_API_DELETE_PARSER    = 2,

    /* index export/import */
    PARSER_API_INITIALIZE_INDEX = 10,
    PARSER_API_IMPORT_INDEX     = 11,
    PARSER_API_EXPORT_INDEX     = 12,

    /* movie properties */
    PARSER_API_IS_MOVIE_SEEKABLE = 20,
    PARSER_API_GET_MOVIE_DURATION = 21,
    PARSER_API_GET_USER_DATA     = 22,

    PARSER_API_GET_NUM_TRACKS    = 25,
```

```
PARSER_API_GET_NUM_PROGRAMS    = 26,  
PARSER_API_GET_PROGRAM_TRACKS  = 27,  
  
/* generic track properties */  
PARSER_API_GET_TRACK_TYPE      = 30,  
PARSER_API_GET_TRACK_DURATION  = 31,  
PARSER_API_GET_LANGUAGE       = 32,  
PARSER_API_GET_BITRATE        = 36,  
PARSER_API_GET_DECODER_SPECIFIC_INFO    = 37,  
  
/* video properties */  
PARSER_API_GET_VIDEO_FRAME_WIDTH    = 50,  
PARSER_API_GET_VIDEO_FRAME_HEIGHT    = 51,  
PARSER_API_GET_VIDEO_FRAME_RATE      = 52,  
  
/* audio properties */  
PARSER_API_GET_AUDIO_NUM_CHANNELS    = 60,  
PARSER_API_GET_AUDIO_SAMPLE_RATE     = 61,  
PARSER_API_GET_AUDIO_BITS_PER_SAMPLE = 62,  
  
PARSER_API_GET_AUDIO_BLOCK_ALIGN     = 65,  
PARSER_API_GET_AUDIO_CHANNEL_MASK    = 66,  
PARSER_API_GET_AUDIO_BITS_PER_FRAME  = 67,  
  
/* text/subtitle properties */  
PARSER_API_GET_TEXT_TRACK_WIDTH = 80,  
PARSER_API_GET_TEXT_TRACK_HEIGHT = 81,  
  
/* sample reading, seek & trick mode */  
PARSER_API_GET_READ_MODE = 100,  
PARSER_API_SET_READ_MODE = 101,  
  
PARSER_API_ENABLE_TRACK = 105,  
  
PARSER_API_GET_NEXT_SAMPLE = 110,  
PARSER_API_GET_NEXT_SYNC_SAMPLE = 111,  
  
PARSER_API_GET_FILE_NEXT_SAMPLE = 115,  
PARSER_API_GET_FILE_NEXT_SYNC_SAMPLE = 116,  
  
PARSER_API_SEEK = 120  
};
```

3.3. Handle of Core Parser

```
typedef void * FsIFileHandle;
```

3.4. Other Constants

3.4.1. Media Types

```
typedef enum
{
    MEDIA_TYPE_UNKNOWN = 0,
    MEDIA_VIDEO,
    MEDIA_AUDIO,
    MEDIA_TEXT, /* subtitle text or stand-alone application, string-based or bitmap-based */
    MEDIA_MIDI
}MediaType; /* Media types of a track.*/
```

Please see the common header file “fsl_media_types.h” for the media and decoder type definitions.

3.4.2. User data ID

```
typedef enum
{
    USER_DATA_TITLE = 0,
    USER_DATA_LANGUAGE, /* user data may tell the language of the movie as a string */
    USER_DATA_GENRE,
    USER_DATA_ARTIST,
    USER_DATA_COPYRIGHT,
    USER_DATA_COMMENTS,
    USER_DATA_CREATION_DATE,
    USER_DATA_RATING,
    USER_DATA_ALBUM,
    USER_DATA_VCODECNAME
    USER_DATA_ACODECNAME
}UserDataID;
```

4. Callback Functions

4.1. File I/O callback

```
typedef struct _FslFileStream
{
    FslFileHandle (*Open)(const uint8 * fileName, const uint8 * mode, void * context);
    int32 (*Close)(FslFileHandle handle, void * context); /* Close the stream */
    uint32 (*Read)(FslFileHandle handle, void * buffer, uint32 size, void * context);
    int32 (*Seek)(FslFileHandle handle, int64 offset, int32 whence, void * context);
    int64 (*Tell)(FslFileHandle handle, void * context);
    int64 (*Size)(FslFileHandle handle, void * context);
    int64 (*CheckAvailableBytes)(FslFileHandle handle, int64 bytesRequested, void * context);
} FslFileStream;
```

Description:

file I/O interface on a file or live source.

open

Open a local file or URL.

Arguments:

fileName [in] File name or url to open.

To open the movie source file, just set file name to NULL.

To open another external file for some track (eg. MP4), set the url.

mode [in] Open mode, same as libc. Such as "rb".

Return value:

Handle of the opened file. NULL for failure.

read

Read data from the file.

Arguments:

handle [in] Handle of the file.

buffer [in] Pointer to a block of memory, to receive the data.

size[in] Data size to read, in bytes.

Return value:

The total number of bytes successfully read.

If this number differs from the size parameter, either an error occurred or the EOF was reached.

seek

Seek the stream.

Arguments:

handle [in] Handle of the file.

offset [in] The offset.

To move to a position before the end-of-file, you need to pass a negative value in offset and set whence to SEEK_END.

whence in] The new position, measured in bytes from the beginning of the file, is obtained by adding offset to the position specified by whence.

SEEK_SET - Set position equal to offset bytes.

SEEK_CUR - Set position to current location plus offset .

SEEK_END - Set position to end-of-file plus offset.

Return value:

Upon success, returns 0; otherwise, returns -1.

tell

Tell the position of the file pointer

Arguments:

handle [in] Handle of the file.

Return value:

Returns the position of the file pointer in bytes; i.e., its offset into the file stream.

If error occurs or this feature can not be supported (eg. broadcast application), returns -1.

size

Tell the size of the entire file.

Arguments:

handle [in] Handle of the file.

Return value:

Returns the file size in bytes.

If error occurs or this feature can not be supported (eg. broadcast application), returns -1.

check_available_bytes

Tell the available bytes of the file, especially useful for a live source file (streaming).

The parser can decide not to read if cached data is not enough and so avoid reading failure in unexpected context.

For a local file, any bytes request from the parser can be met as long as it's within the file range.

Arguments:

handle [in] Handle of the file.

bytes_requested [in] Bytes requested for further parsing. This information can help the application to cache enough data before calling parser API next time.

Return value:

If the file source can always meet the data reading request unless EOF (eg. a local file or a pull-mode live source), returns the data size from the current file pointer to the file end.

Otherwise (eg. a push-mode live source), returns the cached data size.

If error occurs or this feature can not be supported (eg. broadcast application), returns -1.

close

Close the file.

Arguments:

handle [in] Handle of the file.

Return value:

Upon success, returns 0; otherwise, returns -1.

4.2. Memory operation callback

typedef struct

```
{
    void* (*Calloc) (uint32 numElements, uint32 size);
    void* (*Malloc) (uint32 size);
    void (*Free) (void * ptr);
    void* (*ReAlloc)(void * ptr, uint32 size); /* necessary for index scanning!*/
}
```

}ParserMemoryOps; /* callback operation callback table */

4.3. Output buffer Request Callback

typedef struct

```
{
    uint8* (*RequestBuffer) (uint32 streamNum,
                             uint32 *size,
                             void ** bufContext,
                             void * parserContext);

    void (*ReleaseBuffer) (uint32 streamNum,
                           uint8 * pBuffer,
```

```
void * bufContext,  
void * parserContext);
```

```
}ParserOutputBufferOps;
```

Description:

Callback functions to request/release an output buffer.

Usually, the core parser requests an output buffer, fill the media data and return it to the application on API to read next sample by the output argument "sampleBuffer". But, on flushing (eg. on seek or deletion), the core parser need explicitly release all buffers not output yet.

RequestBuffer

Request an output buffer.

Arguments:

streamNum [in] Track number, 0-based.

size [in,out] The requested buffer size as input, and the size actually got as output, both in bytes.

The actually got size can be larger than the requested size, and the parser can make full use of the buffer.

bufContext [out] A buffer context from the application. The parser shall not modify it.

parserContext [in] The parser context from the application, got on parser creation.

Return value:

Buffer pointer. NULL for failure.

ReleaseBuffer

Release an output buffer explicitly.

Arguments:

streamNum [in] Track number, 0-based.

pBuffer [in] Buffer to release.

bufContext [in] The buffer context from the application, got on requestBuffer().

parserContext [in] The parser context from the application, got on parser creation.

Return value: none.

5. API Calling Sequence

This section describes the API Calling Sequence. Without explicitly explanation, the user shall go ahead only if previous step succeeds.

Please refer to the test application code for more details.

- **Open the parser shared library, and query its interface to get all implemented function pointers.**

FslParserQueryInterface

- **Check core parser version (optional)**

ParserVersionInfo()

- **Create the parser**

CreateParser()

As long as the parser is created successfully, it must be deleted as the final step to free the resources.

- **Initialize index table**

(a) For the 1st play, initialize the index table by loading it from the movie file. It can be time consuming for a long movie.

InitializeIndex()

And then export the index table to the external database, for fast opening it next time (optional)

ExportIndex()

(b) For a movie not playing for the 1st time, if its index has be exported before, import the index table from the external database,

ImportIndex()

It's much faster than loading again from the file. Of course, you call still choose to use *InitializeIndex()*.

NOTE: If the index table loading fails, normal playback will not be affected. The user can still seek to the beginning of the movie and start reading A/V samples.

- **Get properties of the movie and tracks, as well as the user data**

IsSeekable()

GetMovieDuration

GetNumTracks() or *GetNumPrograms()*, *GetProgramTracks()*

...

GetTrackType

...

GetVideoFrameWidth

GetVideoFrameHeight

GetAudioNumChannels

GetAudioSampleRate

...

- **Enable wanted tracks and disable unwanted tracks**

EnableTrack

- **Check the the reading mode and change the mode if possible**

GetReadMode

SetReadMode

- **Seek to the beginning of the movie**

Must perform a seeking on all enabled tracks, with target time 0 us.

Seek()

Of course, seek to any time within the play duration is reasonable.

- **Reading samples for playback**

(a) For normal playback (rate = 1X), begin to read A/V samples one by one sequentially from the selected tracks. The user chooses which tracks to read.

GetNextSample() or *GetFileNextSample*

If only part of sample is got due to buffer size restriction, repetitively calling the same function until all sample data are got.

(b) For trick mode (FF/RW), can only pick sync sample (video key frames)

GetSyncSample(direction) or *GetFileNextSyncSample*

If only part of sample is got due to buffer size restriction, repetitively calling the same function until all sample data are got.

(c) A seeking can be performed during the playback. *Seek()*

- **Delete the parser**

DeleteParser()

