



User Guide

2.1

OpenCube Technologies SAS

10 Avenue de l'Europe
Parc Technologique du Canal
31520 Ramonville St Agne – FRANCE

Tel : +33 (0)561 285 606
Fax : +33 (0)561 285 635
E-mail : contact@opencubetech.com
MXFTk Support : support_mxftk@opencubetech.com
Internet : <http://www.opencubetech.com>

User Guide version 9.0 of MXFTk Application Programming Interface version 2.1.

Copyright © 2007 OpenCube Technologies SAS.

Software and user guides described in this document are protected by Copyright.

No reproduction, distribution or use in whole or in part of any content is permitted without prior authorization of OpenCube Technologies SAS.

Any use, for any purpose, not allowed in the terms of the license is strictly forbidden.

OpenCube Technologies SAS uses reasonable efforts to include accurate, complete and current information in this document, however, OpenCube Technologies SAS does not warrant that the content herein is accurate, complete, current, or free of technical or typographical errors. OpenCube Technologies SAS reserves the right to make changes and updates to any information contained within this document without prior notice.

OpenCube Technologies SAS shall not be responsible for any errors or omissions contained in this document, and in particular OpenCube Technologies SAS shall not be liable for special, indirect, consequential, or incidental damages, or damages for lost profits, loss of revenue, or loss of use, arising out of or related to the information contained in this document, whether such damages arise in contract, negligence, tort, under statute, in equity, at law or otherwise.

[Linux is a registered trademark of Linus Torwald]

[Microsoft, MS DOS and Windows are registered trademarks of Microsoft Corporation.]

[Mac OS X is a registered trademark of Apple Computer, Inc.]

[CodeWarrior is a registered trademark of Metrowerks, a Motorola company.]

[C Builder is a registered trademark of Borland Software Corporation.]

[eVTR and XDCam are registered trademarks of Sony Corporation.]

[P2 is a registered trademark of Panasonic Corporation.]

[K2 is a registered trademark of Thomson Grass Valley Corporation.]

[Spectrum is a registered trademark of Omneon Corporation.]

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

All other trademarks and copyrights are the properties of their respective owners.

Table of Contents

TABLE OF CONTENTS	3
1. PRELIMINARIES	8
1.1 MXF File Format	8
1.2 MXFTk.....	8
1.3 MXFTk Overview	9
1.3.1 MXF File Reading.....	10
1.3.2 MXF File Writing	10
1.3.3 MXF File Update	11
1.3.4 Partial Restore	11
1.3.5 External References	11
1.3.6 MXFTk Memory Management	11
1.3.7 MXFTk Error Handling	12
1.3.8 Windows Compilation	12
1.3.9 A word about Unicode	12
1.4 Installation Procedure	12
1.4.1 Linux Platform	12
1.4.2 Mac OS X platform.....	13
1.4.3 Windows platform.....	13
1.5 Deployment Procedure	14
1.5.1 Linux Platform	14
1.5.2 Mac OS X platform.....	14
1.5.3 Windows platform.....	14
1.6 MXFTk Versions History	15
1.6.1 MXFTk 1.1 New Features.....	15
1.6.2 MXFTk 1.2 New Features.....	15
1.6.3 MXFTk 1.3 New Features.....	15
1.6.4 MXFTk 1.4 New Features.....	16
1.6.5 MXFTk 1.5 New Features.....	16
1.6.6 MXFTk 2.0 New Features.....	16
1.6.7 MXFTk 2.1 New Features.....	16
2. API CONFIGURATION AND ERROR MANAGEMENT	17
2.1 I_mxf_error_handler Class Reference	17
2.2 I_mxf_error Class Reference	18
3. MXF FILES.....	20
3.1 I_mxf_file Class Reference	20

3.2 I_op1a_file Class Reference	25
3.3 I_evtr_file Class Reference.....	27
3.4 I_xdcam_imx_file Class Reference	27
3.5 I_xdcam_dv_file Class Reference	28
3.6 I_xdcam_hd_file Class Reference	29
3.7 I_xdcam_proxy_file Class Reference	29
3.8 I_op1b_file Class Reference	30
3.9 I_op1c_file Class Reference.....	31
3.10 I_op2a_file Class Reference.....	32
3.11 I_op2b_file Class Reference	34
3.12 I_op2c_file Class Reference.....	35
3.13 I_op3a_file Class Reference.....	37
3.14 I_op3b_file Class Reference	38
3.15 I_op3c_file Class Reference.....	40
3.16 I_opatom_file Class Reference	42
3.17 I_dcp1_file Class Reference.....	43
3.18 I_opatom_assembler Class Reference	43
3.19 Panasonic P2 Functions Reference	45
 4. MATERIAL	 47
4.1 I_generic_material Class Reference	47
4.2 I_track Class Reference.....	49
4.2.1 I_track_item Class Reference.....	52
4.2.2 I_source_clip Class Reference	53
4.2.3 I_dm_source_clip Class Reference	55
4.2.4 I_dm_segment Class Reference	57
 5. CONCRETE MATERIAL	 59
5.1 I_concrete_material Class Reference	59
5.2 I_concrete_track Class Reference.....	61
5.3 I_essence_type Class Reference	63

6. METADATA MATERIAL	67
6.1 I_metadata_material Class Reference.....	67
6.2 I_metadata Class Reference	68
6.3 I_property Class Reference	70
6.4 I_value Class Reference	71
6.5 I_umid Class Reference	73
6.6 I_umid64 Class Reference	73
7. TIMECODE MATERIAL	75
7.1 I_timecode_material Class Reference	75
7.2 I_timecode Class Reference.....	76
8. STREAMING	79
8.1 I_essence_stream_task Class Reference	79
8.2 I_crypted_essence_stream_task Class Reference.....	81
8.3 I_input_metadata_stream_task Class Reference	82
8.4 I_output_mxf_stream_task Class Reference	82
8.5 I_input_mxf_stream_task Class Reference.....	83
8.6 I_input_partial_mxf_stream_task Class Reference	85
9. ESSENCE DESCRIPTORS.....	86
9.1 I_mxf_file_descriptor Class Reference.....	86
9.2 I_mxf_generic_picture_essence_descriptor Class Reference	86
9.3 I_mxf_cdc_picture_essence_descriptor Class Reference.....	87
9.4 I_mxf_mpeg2_video_descriptor Class Reference.....	88
9.5 I_mxf_rgba_picture_essence_descriptor Class Reference.....	89
9.6 I_mxf_jpeg2000_picture_subdescriptor Class Reference.....	89
9.7 I_mxf_generic_sound_essence_descriptor Class Reference	90
9.8 I_mxf_wave_audio_essence_descriptor Class Reference.....	91
9.9 I_mxf_aes3_audio_essence_descriptor Class Reference.....	92

9.10 I_mxf_generic_data_essence_descriptor Class Reference	92
10. DATA HANDLING INTERFACES	94
10.1 concrete_list Class Reference	94
10.2 generic_list Class Reference	94
10.3 I_timecode_component Class Reference	95
10.4 locators Class Reference	95
10.5 crypted_locator_element Class Reference	96
10.6 crypted_locators Class Reference	97
10.7 metadata_list Class Reference	97
10.8 ordered_timecode_list Class Reference	98
10.9 ordered_track_item_list Class Reference	98
10.10 properties_list Class Reference	99
10.11 rational Class Reference	99
10.12 track_list Class Reference	100
10.13 mxf_file_list Class Reference	101
11. DCP CREATOR	102
11.1 Composition Playlist	102
11.1.1 I_Marker Class Reference	102
11.1.2 I_MarkerAsset Class Reference	102
11.1.3 I_PictureTrackFileAsset class reference	103
11.1.4 I_SoundTrackFileAsset class reference	105
11.1.5 I_SubtitleTrackFileAsset class reference	106
11.1.6 I_Reel Class Reference	107
11.1.7 I_CompositionPlayList Class Reference	108
11.2 Packing List	109
11.2.1 I_Asset Class Reference	110
11.2.2 I_PackingList Class Reference	110
11.3 Asset Map and Volume Index	111
11.3.1 I_Chunk Class Reference	112
11.3.2 I_Asset Class Reference	112
11.3.3 I_AssetMap Class Reference	113
11.3.4 I_VolumeIndex Class Reference	114
11.4 DCPCreator	114
11.4.1 Options and creation	115
11.4.2 I_DCPCreator class reference	116

MXFTK	118
Classes	118
Typedefs	119
Functions.....	119
Enumeration Types.....	122
Error Messages.....	125
REFERENCES	130

1. Preliminaries

1.1 MXF File Format

MXF (standing for Material eXchange Format) is a file format aiming to improve data and metadata exchange. The targeted objective is the interoperability between content creation mainframes, work stations and peripherals.

This wrapper file format was designed to make use of current and forthcoming data formats. Hence, it not only allows exchange of contents in MPEG, DV or else but also improves interoperability between editing systems. In the mean time, it also permits conveyance of metadata following standardized schemes.

1.2 MXFTk

MXFTk is a C++ library for Linux, Mac OS X and Windows operating systems that enables creation and reading of MXF files following the MXF norm from the SMPTE. The use of this library is performed thanks to a set of classes and methods that constitutes an API (Application Programming Interface). The main objective of MXFTk is to let developers manipulate MXF files at will without prior knowledge of the complex mechanisms of this format. Nonetheless, a basic understanding of the MXF file structure is mandatory to build and use these files. API users referring to the MXF File Format glossary [377M] will find definitions specific to the MXF terminology. In order to get a first insight over these requisites, it is advised to read the MXF Engineering Guideline [EG41]. Metadata management also requires proficiency with metadata schemes covered by the SMPTE MXF Descriptive Metadata Engineering Guideline [EG42] and the SMPTE MXF Descriptive Metadata Scheme [380M].

The MXF file format manipulates two distinguished types of metadata: structural and descriptive. Structural metadata builds the architecture of an MXF file: it describes the structure and the ordering of the file content. On the other hand, descriptive metadata corresponds to the metadata set by the user (DMS1 or else).

The API MXFTk is built so that the descriptive metadata is freely and fully filled by the API user while the structural metadata remains entirely controlled by the API. Therefore the API task lies in the conversion of C++ objects into structural metadata and vice-versa.

The following table summarizes the main features of MXFTk:

TAB. 1: MXFTk versions.

	MXFTk
Op1a to Op3c files	Read/Write/Partial Restore
OpAtom files	Read/Write/Partial Restore
SONY XDCam files	Read/Write/Partial Restore
SONY eVTR files	Read/Write/Partial Restore
Panasonic P2 files	Read/Write/Partial Restore
Omneon Mediagrid files	Read/Write/Partial Restore
Thomson K2/Infinity files	Read/Write/Partial Restore
External References	Supported
IMX, MPEG Long-GOP, DV, DVCPro	Supported
Uncompressed Images, JPEG2K, VC-3(DnxHD)	Supported
BWF, AES, 8-channel AES, A-law	Supported
Active/Passive Streaming	Supported
DCI Package Creation and AES Encryption	Supported
Descriptive Metadata	Read/Write/Update
Timecode	Read/Write/Update
Random Access	Supported
Partitioning	Supported
Index Tables	Supported
Reverse Play	Supported
Platforms	Windows, Linux, Macintosh PPC & i386

1.3 MXFTk Overview

MXFTk library relies on three fundamental concepts:

- MXF files.
- Audiovisual, metadata and timecode materials.
- Tracks.

These three models are organized as follows:

An MXF file includes a given number of materials: more precisely, it holds at least one output material and one embedded material. Hence, the simplest files embed no more than one audiovisual material (for instance a DV clip) that is also the output material (it is played as is; no editing of the source is performed). More intricate files may perform an edition by cutting and mixing several audiovisual sources. It is even conceivable to create MXF files containing a large number of materials and offering various output editions.

A material represents data. Four kinds of materials can be manipulated with the API MXFTk: audiovisual material (*concrete_material*), *metadata material*, time positioning material (*timecode_material*) and editing material (*generic_material*). For instance, let's consider a material embodying a multimedia file (an MPEG clip) containing both video and audio data to be played for a given duration. In order to identify its content, the material carries an enumeration of tracks (in our simple case, there will be one audio track and one video track in the material).

As stated just before, tracks represent the content of the materials. Each of them holds a specific type of data (audio, video, etc.) as well as its time positioning and duration. The tracks of a *generic_material* can even contain references to *concrete_materials* allowing editing of the audiovisual sources. This would be the case of a *generic_material* aiming to play sequentially two video clips; each clip would be stored in the video track of a *concrete_material*. The two resulting materials would then be referenced one after the other by the output *generic_material*'s tracks to get the desired file play out.

MXF files complexity is depicted by their Operational Pattern (also called OP in the MXF terminology). Nine operational patterns ranging from the simplest (Op1a) to the trickiest (Op3c) can be distinguished as stated by the following table. Their precise and exhaustive description can be found in the documents [EG41] and [377M]. Another particular operational pattern is called OpAtom. MXF files following this pattern contain a single embedded audiovisual material but still allow complex editing by referencing material from other OpAtom files.

TAB. 2: Operational Pattern

		Complexity		
		single	play-list	edit
	Single	OP1a	OP2a	OP3a
Material	Ganged	OP1b	OP2b	OP3b
	Alternate	OP1c	OP2c	OP3c

- *Op1a*: the simplest and commonest *operational pattern* containing a single *concrete_material* which is played as is by the unique *generic_material*. Files produced by the eVTR from Sony are an extension of the *Op1a*.
- *Op2a*: there is a single *generic_material* consisting in several *concrete_materials* played sequentially.
- *Op3a*: there is a single *generic_material* consisting in playing sequentially some clips cut from the *concrete_materials*.
- *Op1b*: there is a single *generic_material* playing simultaneously the *concrete_materials*.
- *Op2b*: this operational pattern extends the *Op1b* just as the *Op2a* was extending the *Op1a*: it connects sequentially *concrete_materials* played simultaneously.
- *Op3b*: this operational pattern extends the *Op2b* just as the *Op3a* was extending the *Op2a*: it connects sequentially clips cut from *concrete_materials* played simultaneously.
- *Op1c*: there is no restriction on the number of *generic_materials*. However, each of them plays simultaneously one or more *concrete_materials*.
- *Op2c*: there is no restriction on the number of *generic_materials*. However at least one of them corresponds to an *Op2b material package*.

- *Op3c*: the most intricate operational pattern. It has no restriction on the number of *generic_materials*. However, at least one of them corresponds to an *Op3b generic_material*.
- *OpAtom*: this *operational pattern* embeds a single *source package* which consists in a single source (only video or audio). However, the *generic_material* may allow building complex editing (*Op1a*, *Op1b*, *Op2a*, and *Op2b*) by referencing *concrete_materials* from other *OpAtom* files. This pattern is heavily supported by manufacturers such as Panasonic.

MXFTk will let you read and write *Op1a-3c* and *OpAtom* files.

1.3.1 MXF File Reading

The reading of an MXF file begins with the instantiation of an *mxf_file* object with the complete path to the file to unwrap. The version 2.0 introduced several opening modes optimized depending on the task you want to perform with the MXF file (read metadata only, access to the essence, random access to the essence). This creation of the *mxf_file* object automatically launches the decoding process. Upon completion the number of output materials (*generic_material*) and embedded materials (*concrete_material*) can be retrieved from that object. These materials can be questioned to find out which decoders are required to decipher the materials' tracks content. Finally, the API grants access to the binary data through reading methods. This scheme can be repeated whenever the user wishes to read or play the audiovisual stream of a file.

Similarly, the descriptive metadata can be read as follows. The first step consists in checking the availability of metadata tracks in each material (this would be an **I_track** object which type is *timeline_metadata*, *event_metadata* or *static_metadata*). In the next step, the *metadata_materials* held by *dm_segments* are retrieved from these metadata tracks. To end with, each *metadata_material* includes a *metadata* object conveying the descriptive metadata. The *metadata* object is the root of an arborescence structure whose nodes contain properties (**I_property**). Each property can be read separately. MXFTk also provide methods to directly output the metadata in an XML file or buffer.

1.3.2 MXF File Writing

Depending on the intricacy of the file to be written, the user chooses the operational pattern that best matches its purpose. For instance, no editing will be allowed in a straightforward *Op1a* pattern while imbricate materials and a wide range of file play outs will become possible in an *Op3c* pattern. Once chosen, the appropriate object deriving from *mxf_file* can be instantiated (an *op1a_file* for instance). Then, for each multimedia source (essence in the MXF terminology) to embed in the file, a *concrete_material* must be created. These materials must be appended to the *mxf_file* just built using the API methods available for this operational pattern. Next it is also possible to configure several properties of the file such as partitioning, index table creation, header metadata repetition, etc. Finally the function **I_mxf_file::flush()** will launch the creation process.

Descriptive metadata can be used to annotate output or embedded materials, either in their entirety or only during a limited time scope. MXFTk leaves the option to perform an edition of the descriptive metadata, stating for instance the new actors appearing while playing a video clip. The descriptive metadata creation process is subdivided in several steps. First of all a new metadata track must be created. Depending on the properties of the metadata to be added, the user must opt for the most appropriate kind of metadata track. If the metadata annotates the content of the material in a time-linear fashion, then a *timeline_metadata* track is required. On the other hand, if the metadata documents the material with a set of non-adjacent or overlapping events, then an *event_metadata* track suits. Finally, if the metadata does not contain time references and documents the material's tracks as a whole, the user should pick a *static_metadata* track.

Let's consider the simple case where MXFTk user wishes to annotate the output material of an *Op1a* MXF file. To do so, he will be required to go through the following steps:

- Create a metadata tree **I_metadata** from a valid XML file or step-by-step using the API methods.
- Link the resulting metadata tree to a new metadata material **I_metadata_material**.
- Create a new *static_metadata* track **I_track** in the **I_generic_material** to annotate.
- Append the **I_metadata_material** to the newly created track.

1.3.3 MXF File Update

When updating an MXF file the user may change the timecode and add or remove descriptive metadata. In order to do so, an *mxfile* interface is created, opening the MXF file to be updated. Using the appropriate interfaces **I_timecode_material** and **I_metadata_material** you get access to the function performing timecode and metadata update. Note that the changes made to the file will only be performed when closing it using the function **UpdateAndFreeMxfFileInterface**.

Warning! If you update an MXF file from a given manufacturer, there is no guarantee that it will remain compatible with the manufacturer's device.

1.3.4 Partial Restore

MXF file format provides mechanisms to perform partial restore of MXF files. MXFTk lets you perform this task in a single call to a C function. You simply need to specify the source MXF file and the time span you wish to retrieve thanks to the timecode in and out values. The API will create a new object *mxfile* ready to be flushed on disk or streamed in a networked environment. Please refer to the function **NewPartialRestoreMxfFileInterface** from **I_mxf_file** class reference for a complete overview.

1.3.5 External References

In some cases, you will need to decode or create MXF files without any embedded audiovisual material. The source video and audio data will be stored in separate files either as raw media files or as several MXF files. In that case, it will require a little bit more of processing.

- When decoding an MXF file containing external references, the location of the referenced files can be retrieved thanks to the function **I_essence_type::external_references()**. After checking the validity of the references, they can be loaded using **I_concrete_track::set_external_ref_file()**. Then, the *mxfile* object can be manipulated seamlessly as if the audiovisual material was embedded in the file.
- When creating an MXF file containing external references to raw media files, the function **NewExtConcreteMaterialInterface()** will be called in place of **NewConcreteMaterialInterface()** but the creation process will remain similar.
- When creating an MXF file containing external references to other MXF files, **I_op1a_file** to **I_op3c_file** interfaces provide specific interface to append an external *concrete_material* but the creation process will also remain similar.
- When performing the partial restore of an MXF file containing external references, the referenced raw or MXF files will not be involved in the process. It will create a single new file referencing only the selected time span but the referenced files will remain untouched. If the referenced MXF files need to be partly restored as well, a partial restore process should be launched on each of them individually first. Then the *concrete_material* from each newly created file can be assembled in a new MXF file.

1.3.6 MXFTk Memory Management

The C++ classes of the API are pure virtual classes (interfaces) that can not be instantiated with a "new". For that reason, memory allocation and deallocation are handled by "C" style functions whose name usually starts with "New" or "Free". In terms of implementation on the application side, the consequence is that you will exclusively manipulate pointers on MXFTk interfaces. For optimal memory management reasons, any object allocated by the API must be freed using MXFTk memory deallocation methods. Neither explicit nor implicit calls to constructors and destructors via *new*, *delete* or even *dynamic_cast* should occur on MXFTk objects. Failure to meet these requirements will invariably result in memory leaks and/or crashes in your code. For the same reasons, MXFTk will never destroy objects or buffers that were allocated on the library's client side; you will always remain responsible for the deletion of the memory you allocated.

Header files documentation clearly specifies the deletion responsibility for each parameter and returned value.

1.3.7 MXFTk Error Handling

The class **I_mxf_error** helps managing, detecting and identifying MXFTk internal errors. It can be useful to notice, for instance, wrong file paths or unrecognized property labels. When an error arises, the API pushes it on an error stack that can be questioned thanks to an error manager **I_mxf_error_handler**. Raised errors are sorted depending on their gravity and also return an error identification helping to perform an advanced error processing. Finally, the error handler includes support for internationalization by letting you specify the current active language for error messages. MXFTk comes along with an English American dictionary but you may translate the error messages in any language if required.

MXFTk user is encouraged to get used to MXFTk functionalities with the help of the numerous examples accompanying the installation. These programs explore the most common scenarios: an MXF file wrapper, an MXF File Unwrapper, an MXF to XML converter, an OpAtom wrapper, an OpAtom unwrapper, an MXF file wrapper with import of XML metadata, an MXF file wrapper with timecode redefinition, an MXF file wrapper in streaming mode, an MXF file unwrapper in streaming mode, etc. These examples are also particularly useful to apprehend MXFTk error management.

1.3.8 Windows Compilation

MXFTk interfaces do not contain dependencies on the C runtime library. In other words, the dynamically linked library `mxf_tk.dll` should compile with the runtime library of your choice (either static, multithreaded or dll multithreaded). Moreover, to address the maximum number of compilers, import libraries following both the COFF and OMF name mangling rules can be found in the `lib` directory of your installation. You should use the COFF library with Microsoft and Metrowerks compilers while the OMF library is meant to be used with Borland compilers. Your installation directory includes some projects ready to be compiled with Microsoft Visual C++ and Borland C Builder.

1.3.9 A word about Unicode

Unicode UTF16 characters are manipulated thanks to the **wchar_t** data type within MXFTk. However, you must be aware that depending on the platform the size of this data type is different. On the Windows platform `sizeof(wchar_t) = 2` while on the Linux and Mac platform `sizeof(wchar_t) = 4`. When writing portable code, you should not assume that the size of the Unicode **wchar_t** characters returned by MXFTk will always be 2-byte long.

1.4 Installation Procedure

1.4.1 Linux Platform

- Insert your MXFTk installation CD in your player. Go to the directory MXFTk and execute the shell command “`install_mxf_tk_linux.run [installation_directory]`” and “`install_dcpcreator_linux.run [installation_directory]`” if you also wish to use the DCP functions. If the installation directory is not specified MXFTk and DCPCreator will be installed in `/usr/local`.
- The following elements will be created during the installation procedure:
 - `prefix_path/bin`
 - `prefix_path/lib/libmxf_tk.so.2.0.0`
 - `prefix_path/lib/libmxf_tk.so.2.0`
 - `prefix_path/lib/libmxf_tk.so.2`
 - `prefix_path/lib/libmxf_tk.so`
 - `prefix_path/include/mxf_tk`
 - `prefix_path/share/mxf_tk`
- And for the DCPCreator library:

- prefix_path/bin
 - prefix_path/lib/libdcpcreator.so.1.0.0
 - prefix_path/lib/libdcpcreator.so.1.0
 - prefix_path/lib/libdcpcreator.so.1
 - prefix_path/lib/libdcpcreator.so
 - prefix_path/include/dcpcreator
 - prefix_path/share/dcpcreator
- In the directories called “prefix_path/include/mxf_tk” and “prefix_path/include/dcpcreator” you will find all the header files required to use the API. The directory “prefix_path/share/mxftk/dic” contains the default XML schema (dictionary) loaded while using the API. You will be able to get an electronic version of this user guide browsing down to the directory “prefix_path/share/doc”. Finally, you can remove MXFTk by executing the script “prefix_path/share/bin/unsintall_mxftk.run” and DCPCreator with “prefix_path/share/bin/uninstall_dcpcreator.run”.
 - The libraries require one environment variable to work properly (MXF_HOME). MXF_HOME should point to the installation directory (for instance “prefix_path”) and LD_LIBRARY_PATH is the directory search list used by your linker. You may have to update it to include \$MXF_HOME/lib. It is possible to run MXFTk without setting the MXF_HOME environment variable, for more information on this possibility please refer to the chapter “Deployment Procedure”.
 - You will need a valid license key file in order to use MXFTk. You can get a license key by registering to www.mxftk.com and then you simply need to load the license file you will receive with OpenCube’s license keys manager application. This product is part of your installation package.
 - After completing the previous steps, if you browse down to MXFTk/linux/bin, you will find a file called opencube.mxftk.license.lcs containing the licensing information for your distribution of the API. It usually includes the name of your company and should always remain in the bin folder.
 - Please be aware that if you wish to move the library libmxf_tk.so to another folder, you must necessarily move the license file and update your environment variable accordingly.
 - The “examples” directory contains several shell executables ready to be compiled.

1.4.2 Mac OS X platform

- Insert your MXFTk installation CD in your player and execute MXFTk and DCPCreator MacOSX packages.
- Two new frameworks will be installed in the directory “/Library/Frameworks”. The names of the framework are mxf_tk.framework and DCPCreator.framework.
- In the installation directory you will find all the binary and header files required to use the API. The directory “dic” contains the default XML schema (dictionary) loaded while using the API. Finally you will be able to get an electronic version of this user guide browsing down to the directory “doc”.
- The libraries require one environment variable to work properly (MXF_HOME). MXF_HOME should point to your installation directory. If required, it is possible to run MXFTk without setting the MXF_HOME environment variable, for more information on this possibility please refer to the chapter “Deployment Procedure”.
- You will need a valid license key file in order to use MXFTk. You can get a license key by registering to www.mxftk.com and then you simply need to load the license file you will receive with OpenCube’s license keys manager.
- After completing the previous steps, you will find in your installation directory a file called opencube.mxftk.license.lcs containing the licensing information for your distribution of the API. It usually includes the name of your company if you and should always remain in the bin folder.

1.4.3 Windows platform

- Make sure to have administrator rights before performing this installation.
- Note that during the installation, antivirus software may warn you about the activity of shell commands. These commands are required to perform the installation and you should ignore the antivirus warnings.
- Insert your MXFTk installation CD in your player. The installation procedure should start automatically. If not simply launch Setup.exe.
- Follow the installation procedure. The application “KeyManager” will be installed, from there you will be able

to launch the installation of MXFTk.

- In the installation directory you will find all the binary and header files required to use the API. The directory “dic” contains the default XML schema (dictionary) loaded while using the API. Finally, you will be able to get an electronic version of this user guide browsing down to the directory “doc”.
- The library requires one environment variable to work properly (MXF_HOME). MXF_HOME should point to the installation directory (for instance /Projects/Dlls/MXFTk). You can check that this variable was correctly set during the installation. It is possible to run MXFTk without setting the MXF_HOME environment variable, for more information on this possibility please refer to the chapter “Deployment Procedure”.
- Additionally, the path to mxftk.dll should be appended to your variable PATH to permit dynamic linkage at runtime. This task is also automatically performed during the installation.
- You will need a valid license key file in order to use MXFTk. You can get a license key by registering to www.mxftk.com and then you simply need to load the license file you will receive with OpenCube’s license keys manager.
- The directory MXFTk\lib contains two import libraries respectively in the COFF and OMF format. You should use the COFF library if you compile with Microsoft Visual or Metrowerks CodeWarrior compilers. Alternatively, the OMF library targets users of Borland C Builder compilers.
- The “examples” directory contains several examples ready to be compiled.

1.5 Deployment Procedure

This section describes the requirements to use MXFTk while using and installing it in a third-party application.

1.5.1 Linux Platform

- The following exhaustive list summarizes the files from your MXFTk installation that are mandatory for a correct behavior of MFTk. All these files should be included while deploying MXFTk:
 - prefix_path/bin/opencube.mxftk.license.lcs.
 - prefix_path/lib/libmxftk.so.2.0.0 (and sym links)
 - prefix_path/lib/libdcpcreator.so.1.0.0 (and sym links)
- MXF_HOME should be set during the installation and should point to the directory containing your installation of MXFTk. In your program, you will need to call the function **InitMXF_TK()**.
OR
- MXF_HOME is not set during the installation. In your program you will need to call **InitMXF_TK2()**. The parameter for this function should be the path to the directory containing your installation of MXFTk.

1.5.2 Mac OS X platform

- mxftk.framework and DCPCreator.framework (for DCP) are required in deployment. It is possible to remove any component of these frameworks. Removable components:
 - mxftk.framework/Headers
 - mxftk.framework/Versions/A/Resources/doc
 - mxftk.framework/Versions/A/Resources/examples
 - mxftk.framework/Versions/A/Resources/dic
 - mxftk.framework/Versions/A/include
 - mxftk.framework/Versions/A/Headers
- MXF_HOME should be set during the installation and should point to the directory containing your installation of MXFTk. In your program, you will need to call the function **InitMXF_TK()**.
OR
- MXF_HOME is not set during the installation. In your program you will need to call **InitMXF_TK2()**. The parameter for this function should be the path to the directory containing your installation of MXFTk.

1.5.3 Windows platform

- The following exhaustive list summarizes the files from your MXFTk installation that are mandatory for a correct behaviour of MXFTk. All these files should be included while deploying MXFTk:
 - mxftk.dll
 - opencube.mxftk.license.lcs

- dcpcreator.dll (only required if you use the DCP creation functions)
- msvcp71.dll (in Windows system directory)
- msucr71.dll (in Windows system directory)
- Note that the license file should be located next to mxftk.dll
- MXF_HOME should be set during the installation and should point to the directory containing your installation of MXFTk. In your program, you will need to call the function **InitMXF_TK()**.
OR
- MXF_HOME is not set during the installation. In your program you will need to call **InitMXF_TK2()**. The parameter for this function should be the path to the directory containing your installation of MXFTk.

1.6 MXFTk Versions History

1.6.1 MXFTk 1.1 New Features

Here is a brief summary of the new features of MXFTk 1.1:

- Improved compatibility with all the compilers. Removed all dependencies to the C runtime library.
- Improved Error Management with error identification, classification and error messages internationalization.
- Added complete streaming capability upon wrapping/unwrapping.
- Added support of OpAtom 1-2/a-b files (Advanced version only).
- Added random access to audiovisual data from the generic material (material package) and concrete material (top level source package) (Advanced version only).
- Added access to data from a given timecode value (Advanced version only).
- Added complete support of index tables (Advanced version only).
- New essences now fully supported: AES and MPEG 2 Transport Stream and Program Stream.
- Added a new class identifying the video/audio format of a track.
- Improved management of generic containers.
- Improved installation procedure on Windows platform.
- Optimized file processing.
- Improved support of large files (> 4 Gbytes).

1.6.2 MXFTk 1.2 New Features

Here is a brief summary of the new features of MXFTk 1.2:

- Added support of Panasonic P2 files.
- Added support of SONY XDCam files.
- Improved data retrieval from the essence descriptors (**I_essence_type**).
- New essences now fully supported: A-law and MPEG 4.
- Added support of the Reverse Play feature.

1.6.3 MXFTk 1.3 New Features

Here is a brief summary of the new features of MXFTk 1.3:

- Added the partial restore capability (between two timecode values).
- Improved support of Panasonic P2 files (NTSC).
- Improved support of SONY XDCam files (possibility to retrieve the embedded XML file). See function **I_mxf_file::get_embedded_xml()**.
- Clip wrapping of files larger than 2GB is now enabled.
- The streaming interfaces can now be set as “seekable”.
- New functions for easier timecode manipulation.

1.6.4 MXFTk 1.4 New Features

Here is a brief summary of the new features of MXFTk 1.4:

- Support of MAC OS X platform.
- Added support of operational patterns 1b, 2a and 2b.
- Improved file update (changes are now directly applied to the source).
- Improved descriptive metadata manipulation (new functions to remove existing metadata)
- Panasonic P2 files can now be created with streamed essence sources.
- Introduction of active and passive mode while wrapping MXF files in a streaming environment.
- Mono and stereo AES3 sources embedded in an MXF file can now be extracted in WAVE format.
- Wrapping of SONY XDCam DV and Panasonic P2 files can now be performed with AES3 or WAV sources.

1.6.5 MXFTk 1.5 New Features

Here is a brief summary of the new features of MXFTk 1.5:

- Support of uncompressed images.
- Support of JPEG2K images.
- Support of AES encryption.
- Added compatibility with DCI (Digital Cinema Initiative).
- Added new streaming capabilities.
- Added new metadata dictionary following the latest norm.
- Added a new function to the I_property class helping the creation of metadata trees.

1.6.6 MXFTk 2.0 New Features

Here is a brief summary of the new features of MXFTk 2.0:

- Support of complete Digital Cinema Package creation.
- Support of external references to raw media.
- Support of external references to MXF files.
- Support of Omneon Mediagrid files with external references.
- Support of Thomson K2 files.
- Improved support of Panasonic P2 DVCProHD files.
- Added support of DNxHD.
- Added support of AIFF audio.
- D10 MXF files can now be wrapped/unwrapped using WAVE audio instead of 8-channel AES.
- Mono and stereo Wave files can now be wrapped as AES audio.
- Added new progress function during MXF wrapping.
- Added new MXF files opening modes.
- Simplified deployment procedure.

1.6.7 MXFTk 2.1 New Features

Here is a brief summary of the new features of MXFTk 2.1:

- Support of XDCam HD wrapping.

2. API Configuration and Error Management

void ExtractAesToWave (bool)

When the parameter is set to true, AES3 sources embedded in an MXF file will be extracted in a WAVE file. Default value is *false*.

void ExtractAes8ToWave (bool)

When the parameter is set to true, 8-channel AES3 sources (D10 audio) embedded in an MXF file will be extracted in a Wave file. Note that only the tracks flagged as valid will be extracted. Default value is *false*.

void WrapWaveAsAes (bool)

When the parameter is set to true, mono and stereo Wave files will be embedded as AES sources during the next wrapping process. Default value is *false*.

void WrapWaveAsAes 8(bool)

When the parameter is set to true, Wave files (mono to 8-channel) will be embedded as 8-channel AES sources (D10 audio) during the next wrapping process. Default value is *false*.

void ExtractCustomByKLV (bool)

Use this function to set the essence extraction behaviour when performing MXF unwrapping of custom wrapped MXF files (i.e. XDCam proxy). When the parameter is set to *false* (default) the essence is extracted edit unit by edit unit. When the parameter is set to true, the extraction is performed KLV by KLV and each KLV will contain several edit units. The KLV mode is likely to improve performances.

void InitMXF_TK ()

This function initializes the API MXFTk. Any binary making use of the library should always call it first of all. It prepares the API to be ready to perform future tasks. This function will also validate your license key and the environment variable MXF_HOME should be correctly set before calling this function. Refer to the “Deployment Procedure” section for more information.

void InitMXF_TK2 (const char*)

This function initializes the API MXFTk. Any binary making use of the library should always call this function or the previous one. It prepares the API to be ready to perform future tasks. This function will also validate your license key. The parameter is the complete path toward the directory containing the MXFTk library. Refer to the “Deployment Procedure” section for more information.

void SetDictionary (const char*)

This function specifies a new metadata dictionary (XML scheme) to be used in the following calls of the API. The default dictionary is set upon calling of **InitMXF_TK()** or **InitMXF_TK2()** so you will need to call this function only if you wish to make use of an alternate scheme.

bool GetMxfErrorHandlerInterface (I_mxf_error_handler **)

This function lets you retrieve the interface handling MXFTk errors. This object is created upon loading of the dynamic library and will be shared by all the processes. Thus, this function does not allocate a new **I_mxf_error_handler** and the pointer returned is automatically deleted upon unloading of the dynamic library.

2.1 I_mxf_error_handler Class Reference

```
#include <I_mxf_error.hpp>
```

This class manages a stack of errors for MXFTk. When an error occurs it is added to the stack waiting for the user to retrieve it. It can be cleared once the errors have been retrieved. This class also provides support for internationalization of error messages. Details on how to properly use it and build a **try...catch**-like mechanism can be found in the examples of your installation directory.

Public Member Functions

bool clear ()
bool free_errors (error_list *) const
error_list *get_errors ()
bool has_an_error () const
bool set_error_dictionary (const char *)

Member Functions Documentation

bool clear ()

Removes all the errors from the stack. The stack will be empty after calling this function, ready to enlist future errors. Return *false* if an error occurred.

bool free_errors (error_list *) const

Frees the list of errors returned by **get_errors()**. It only frees the list, errors remain in the stack. To remove errors from the stack use **clear()** instead. Returns *false* if an error occurred.

error_list *get_errors ()

Returns the list of errors currently stored in the stack. Return an empty list if the stack is empty and a NULL pointer if the list of errors could not be retrieved.

bool has_an_error () const

Returns *false* if the error stack is empty and *true* otherwise.

bool set_error_dictionary (const char *)

The default dictionary is in English but if you wish to internationalize your application you may need to define dictionaries for other languages. Call this function to set your own error dictionary. The parameter must define the path toward the new dictionary.

Error dictionaries must be saved in Unicode UTF16 and their syntax is pretty straightforward: there should be one error per line and each line must start with the error id bracketed between two '#' followed by the error message bracketed between '<'>'. For instance:

```
#1002# <Cannot proceed: Trying to add metadata to a picture, sound or data track>
```

Finally your dictionary file should not be larger than 64Kbytes (32K UTF16 characters). The function returns *false* if the new language could not be set. Please note that the complete list of errors can be found at the end of this document.

2.2 I_mxf_error Class Reference

```
#include <I_mxf_error.hpp>
```

This class lets you manipulate and retrieve information from an error that occurred within MXFTk. Errors are managed by the class **I_mxf_error_handler** and are returned by the function **get_errors()**.

Public Member Functions

const wchar_t *get_error_msg () const
gravity get_gravity () const
uint32_t get_id () const

Member Functions Documentation

const wchar_t *get_error_msg () const

Returns a message helping to understand the nature of the error. It is displayed in the language that was set with the function **I_mxf_error_handler::set_language()**. Returns NULL if the error message could not be found.

gravity get_gravity () const

Returns the seriousness of the error. This works as a filter letting you decide what action to undertake depending on the gravity.

uint32_t get_id () const

Returns the unique identifier of the error as stored in the error dictionary.

Related Documentation

enum gravity

Enumeration values:

MXF_NONE
MXF_CAUTION
MXF_WARNING
MXF_FATAL

This enumeration defines the different levels of gravity of an error.

MXF_NONE is returned when the error could not be identified.

MXF_CAUTION is the lowest gravity. This generally means that the user tries to undertake an illegal operation on the MXF file or that an unexpected event occurred. However, this error should not stop MXFTk from working correctly and the validity of the MXF files being manipulated has not been affected by this error. For instance, this gravity of error will be set if you are trying to reach an invalid timecode.

MXF_WARNING denotes a more serious error. The validity of the file being manipulated is at least partially altered and future manipulations of the file may lead to unexpected behaviour. However, it is really likely that the overall behaviour of MXFTk will remain correct while reading or writing other MXF files. For instance, this gravity of error will be set if you are trying to read an invalid MXF file.

MXF_FATAL is the highest gravity. It generally means MXFTk cannot continue to work properly at least with the file being manipulated. For instance, this gravity of error will be set if the toolkit cannot find a valid license key.

3. MXF Files

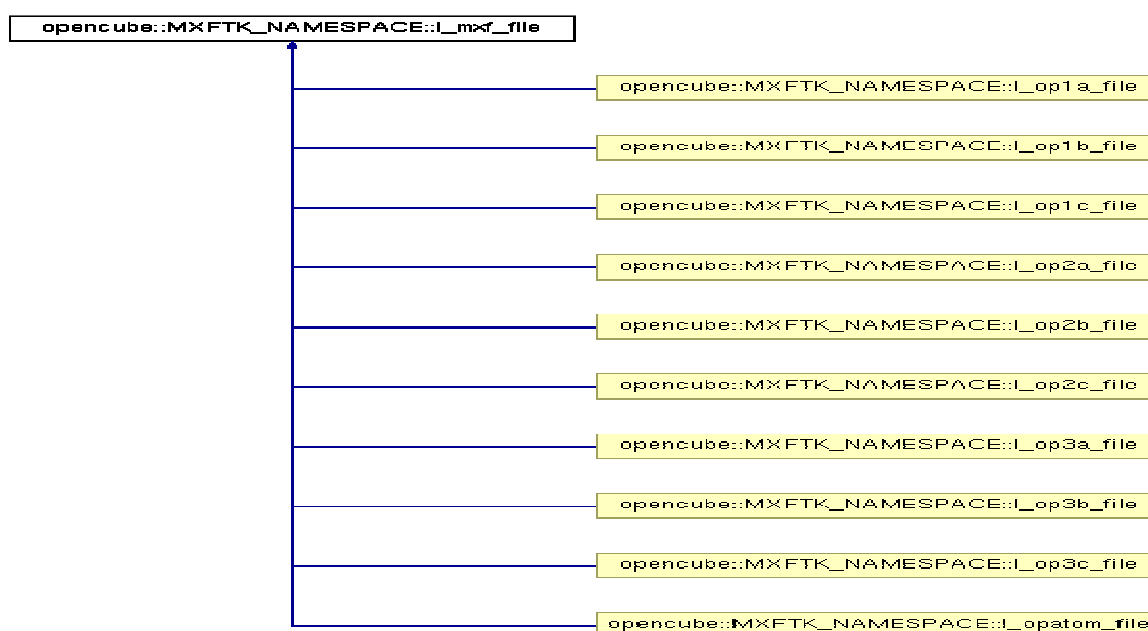
This API release supports Op1a, Op1b, Op1c, Op2a, Op2b, Op2c, Op3a, Op3b, Op3c and OpAtom files. Currently MXFTk relies on C++ classes representing MXF files. Their utility is determined by the task to be performed:

- *mxfile*, to read and update MXF files.
- *op1a_file*, to write Op1a MXF files.
- *op1b_file*, to write Op1b MXF files.
- *op1c_file*, to write Op1c MXF files.
- *op2a_file*, to write Op2a MXF files.
- *op2b_file*, to write Op2b MXF files.
- *op2c_file*, to write Op2c MXF files.
- *op3a_file*, to write Op3a MXF files.
- *op3b_file*, to write Op3b MXF files.
- *op3c_file*, to write Op3c MXF files.
- *evtr_file*, to write MXF files following the same attributes as those generated by the SONY eVTR device.
- *xdcam_imx_file* to write D10 MXF files following the same attributes as those generated by the SONY XDCam camcorder.
- *xdcam_dv_file* to write DVCam+AES MXF files following the same attributes as those generated by the SONY XDCam camcorder.
- *xdcam_proxy_file* to write MPEG4+A-law MXF files following the same attributes as those generated by the SONY XDCam camcorder.
- *opatom_file*, to read and write OpAtom MXF files. Furthermore two C functions called **NewP2Shot** and **NewP2ShotFromStream** generate DVCPro+AES3 MXF files following the same attributes as those generated by the Panasonic P2 camcorder.

3.1 I_mxf_file Class Reference

```
#include <I_mxf_file.hpp>
```

This class should be used whenever you wish to read or update an MXF file.



Public Member Functions

```

concrete_list *embedded_material ()
bool end_of_stream ()
bool flush ()
bool flush_file (const char *)
bool flush_thread_cancel ()
double flush_thread_progress ()
bool flush_stream (I_output_mxf_stream_task *)
bool flush_thread_running ()
bool free_concrete_list (concrete_list *)
bool free_generic_list (generic_list *)
bool free_identifications (metadata_list *) const
metadata_list *get_identifications () const
const char *get_parameter (int) const
const I_metadata *get_preface () const
bool has_external_essence () const
generic_list *low_level_material ()
const char *operational_pattern () const
bool output_embedded_xml (const char*)
generic_list *output_material ()
bool output_xml (const char *)
const unsigned char* output_xml_buffer (uint64_t&)
bool set_parameter (int, const char *)
bool wait_end_flush_thread ()
bool waiting_stream (I_essence_stream_task *&, unsigned int &)
size_t write (const uint8_t *, size_t)

```

Constructor and Destructor Documentation

bool NewMxfFileInterface (I_mxf_file **, const char*, mxf_opening_mode)

Retrieves a pointer on an **I_mxf_file** interface. The second parameter designates the complete path toward the MXF file to be read. The third parameter defines the opening mode. This value can be changed depending on the task to undertake with the file:

- **METADATA_ONLY**: Fastest opening mode but you will only get the first header metadata found while decoding the file. Depending on the file, this header metadata may be open and/or incomplete. The metadata should be considered with "caution". You cannot extract media from the file with this mode.
- **CLOSED_METADATA_ONLY**: Return the best closed header metadata found in the file. The header metadata can be incomplete, that means all the metadata you get will be valid but some of it might be missing. You cannot extract media from the file with this mode.
- **CLOSED_COMPLETE_METADATA**: Return the best header metadata found in the file. This process may be longer than **CLOSED_METADATA_ONLY** but you are sure to get valid metadata. You cannot extract media from the file with this mode.
- **METADATA_AND_LINEAR_PLAYOUT**: Similar to **CLOSED_AND_COMPLETE_METADATA** but also enable extraction of media stored in the file.
- **METADATA_AND_RANDOM_ACCESS**: Similar to **METADATA_AND_LINEAR_PLAYOUT** and also include decoding of index tables enabling faster random access to the media.

The function returns *false* if the allocation failed.

bool NewMxfStreamInterface (I_mxf_file **, I_input_mxf_stream_task *)

Retrieves a pointer on an **I_mxf_file** interface. This function must be called when reading an MXF file received from a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_input_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to retrieve data from the stream. The key functions while opening an **I_mxf_file** with this constructor are **end_of_stream()** and **write()**. Returns *false* if the allocation failed. Please refer to the "Streaming" chapter for a complete overview of the streaming capabilities of

MXFTk.

bool NewPartialRestoreMxfFileInterface (I_mxf_file **, const char*, I_timecode*, I_timecode*)

Retrieves a pointer on an **I_mxf_file** interface and perform a partial restore on the specified MXF file. The second parameter is the complete path to the MXF file to be partially restored. The following timecode values set the timecode in and out defining the time span to be retrieved. This function will create the MXF file in memory, in order to flush it you will need to use one of the **flush()** functions of **I_mxf_file** interface.

This function will retrieve the video, sound and data located between the timecode in (included) and timecode out (also included). These timecode values correspond to the timecode material of the **I_generic_material** of the file. It is not possible to perform a partial restore according to the timecode material of the **I_concrete_material**. Note that if the time span defined by the timecode values is not valid an error will be raised. Possible errors may include an inappropriate frame rate, an invalid drop frame property, etc. Furthermore, if the MXF file contains several **I_generic_material** (several material packages), they will be all partially restored according to the same timecode values. MXFTk will do its best to ensure that the restored file follow the same properties as the original file. It will keep the properties of partitioning, index tables, kag_size, etc.

Returns *false* if an error occurred.

bool FreeMxfFileInterface (I_mxf_file **)

Frees an **I_mxf_file** interface. If the file you provide is an OpAtom file that was parallelized, serialized or synchronized, calling this function will free all the associated OpAtom files so that you need to call this function only once. Returns *false* if the deallocation failed.

bool UpdateAndFreeMxfFileInterface (I_mxf_file **)

Flushes the changes you made to this MXF file (timecode redefinition, metadata update) and frees the **I_mxf_file** interface. If the file you provide is an OpAtom file that was synchronized, calling this function will update and free all the associated OpAtom files so that you need to call this function only once. Returns *false* if the deallocation failed.

Member Functions Documentation

concrete_list *embedded_material ()

Gets a list of **I_concrete_material**. Concrete materials contain all the audiovisual data embedded in the file. According to the MXF terminology they can also be called “Top Level Source Packages”. The list can be freed at any time using **free_concrete_list()**.

bool end_of_stream ()

This function is only useful while reading an MXF file from a stream. It must be called by the user in order to notify MXFTk that the end of the file has been reached and that no more input data will be received from this stream. Returns *false* if an error occurred.

bool flush ()

Causes the creation of the MXF file. This function triggers the on-disk writing of the **I_mxf_file**. The function launches a thread that will perform the writing. Therefore you should not assume that the flush is completed when returning from this function. You should always call the function **wait_end_flush_thread()** to complete the process. It is safer not to perform any changes to the **I_mxf_file** while the flush is still processing.

When an **I_opatom_file** that was assembled thanks to an **I_opatom_assembler** is flushed it is important to understand that all its linked **I_opatom_file** will be flushed at the same time. You need to call this function only once to flush a set of OpAtom files.

Returns *false* if an error occurred.

bool flush_file (const char *)

Similar to the previous function. Forces the flush to occur at the specified location. This function should be called in order to flush a partially restored file.

bool flush_thread_cancel ()

Call this function to force the termination of the thread currently running. Returns *false* if the thread cannot be terminated. Note that the termination is asynchronous.

double flush_thread_progress ()

Returns the current progress of the flush operation. The returned value is comprised between 0.0 (beginning) and 1.0 (end). You should not rely on this function to know if the thread has terminated, instead use the **flush_thread_running()** function.

bool flush_stream (I_output_mxf_stream_task *)

Similar to the previous function. Forces the flush to be launched with the specified streaming interface. This function should be called in order to flush a partially restored file.

bool flush_thread_running ()

Returns *true* if a the thread from the flush function is still running, *false* otherwise.

bool free_concrete_list (concrete_list *)

Frees the list returned by **embedded_material()**. Returns *false* if an error occurred.

bool free_generic_list (generic_list *)

Frees the list returned by **output_material()** or **low_level_material()**. Returns *false* if an error occurred.

bool free_identifications (metadata_list *) const

Frees the list returned by **get_identifications()**. Returns *false* if an error occurred.

metadata_list *get_identifications () const

Gets a list containing the identification metadata of the current file. This structure draws up the list of modifications that occurred on this file throughout its lifetime (a new identification set is generated whenever a file is updated). List's items hold information such as the date of the latest update as well as the name of the product that performed it. Please, refer to the MXF File Format glossary [377M] for further detailed information. You may refer to the function "read_history" from the example "opatom_unwrapper" to get an idea on how to retrieve information from identification sets. The metadata list can be freed at any time using **free_identifications()**.

const char *get_parameter (int) const

Returns the value of the requested parameter (see **set_parameter()** for an overview of the parameters). The string will be set to "unknown" if the value could not be retrieved. The pointer returned must not be destroyed.

const I_metadata *get_preface () const

Returns the Preface set's metadata. Version, embedded essences and descriptive metadata types are properties of this preface descriptor. Please refer to the MXF File Format glossary [377M] for a complete overview of its properties.

bool has_external_essence () const

Returns *true* if the MXF file contains external references to other MXF or media files. This information is computed from the operational pattern value.

generic_list *low_level_material ()

Returns a list of **I_generic_material**. These "low-level" materials are rarely encountered in an MXF file and are not mandatory. They grant access to an informative history of embedded audiovisual material derivation (for instance stating the original files used to build the essence). They may contain some descriptive metadata. According to the MXF terminology, they are also known as "Low Level Source Packages". The list can be freed at any time using **free_generic_list()**.

const char *operational_pattern () const

Returns a string identifying the Operational Pattern of this file. Possible values are "Op1a", "Op1b", "Op1c", "Op2a", "Op2b", "Op2c", "Op3a", "Op3b", "Op3c", "OpAtom_1a", "OpAtom_1b", "OpAtom_2a",

“OpAtom_2b” and “Op_pd_proxy” (for XDcam proxy files). The pointer returned must not be destroyed.

bool output_embedded_xml(const char *)

Returns the XML file embedded within the MXF file (if any). This function is used to retrieve the XML data found in SONY XDcam files. Most of MXF files do not embed this kind of data; therefore in most cases this function will be inoperative. Returns *false* if an error occurred.

generic_list *output_material ()

Gets a list of **I_generic_material**. These materials contain the play out and editing information of this file. Nevertheless, they do not embed audiovisual material but instead they reference the **I_concrete_material** returned by **embedded_material()**. According to the MXF terminology, they are also best known as “Material Packages”. The list can be freed at any time using **free_generic_list()**.

bool output_xml (const char *)

Dumps the content of the current file’s header metadata in an xml file (it includes both descriptive and structural metadata). Returns *false* if an error occurred.

bool output_xml_buffer(uint64_t&)

Dumps the content of the current file’s header metadata in a buffer in XML format and sets its size. The XML buffer includes both descriptive and structural metadata. Returns *NULL* if an error occurred.

bool set_parameter (int, const char *)

This method can be used to adjust advanced MXF parameters. They are briefly described here but their entire specification is covered in the MXF File Format glossary [377M]. The returned value indicates whether the new parameter’s value could be taken into account. Supported parameters are:

1. **KAG_SIZE**, `my_file.set_parameter(KAG_SIZE, "128")`; Sets the KAG (KLV Alignment Grid) size to 128 bytes. This alignment grid can be used to ensure a given spacing between each KLV (Key/Length/Value – elementary constitutive entities of an MXF file). Default value is “512”.
2. **HEADER_REPETITION**, `my_file.set_parameter(HEADER_PARTITION, "2")`; Places a new header metadata every two partitions (even partitions enclose metadata). A nil value (default) only repeats metadata when properties are updated. Default value is “0”.
3. **INDEX_TABLE**, `my_file.set_parameter(INDEX_TABLE, "True")`; Creates an index table for each material of this file. Default value is “False”.
4. **PREFERRED_PARTITION_DURATION**, `my_file.set_parameter(PREFERRED_PARTITION_DURATION, "2.0")`; Tries to adjust the partitions’ size so that their duration remains as close as possible from the specified duration (in seconds). MXFTk can not ensure the accuracy of this size due to structural constraints inherent to the MXF format specification. Besides, a nil duration “0.0” is interpreted as a complete omission of “body partitions”, only a “header” and a “footer” partition will be found in such a file ([377M]). Default value is “0.0”.
5. **REVERSE_PLAY**, `my_file.set_parameter(REVERSE_PLAY, “True”)`; activates the reverse play feature in the file being generated. Default value is “False”.

These parameters have default values, therefore, calls to the function **set_parameter()** remain optional.

bool wait_end_flush_thread ()

This function will only when the flush thread has completed. You should always call this function after flushing an **I_mxf_file**. It will free the memory allocated by the thread and this will also avoid leaving a program while a thread is still running.

bool waiting_stream (I_essence_stream_task *&, unsigned int &)

When performing an MXF wrapping in active streaming mode, you may use this function in order to retrieve the task and stream id that the user should send to MXFTk in order to continue the wrapping process.

size_t write (const uint8_t *, size_t)

This function is only useful while reading an MXF file from a stream. It must be called by the user in order to feed MXFTk with the data received from the streaming device. The first parameter is the buffer containing the data and the second parameter is the size of this buffer. The returned value contains the size effectively written.

Related Documentation

enum

Enumeration members:

KAG_SIZE
HEADER_REPETITION
INDEX_TABLE
PREFERRED_PARTITION_DURATION
REVERSE_PLAY

This enumeration defines all the advanced parameters over which MXFTk grants access.

enum mxf_opening_mode

Enumeration members:

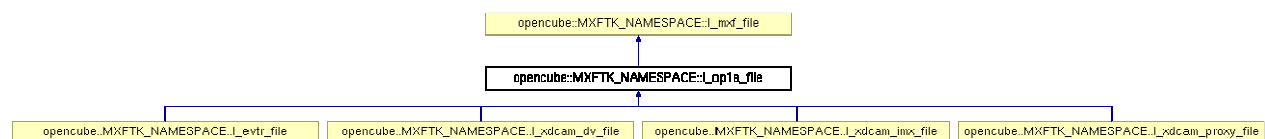
METADATA_ONLY
CLOSED_METADATA_ONLY
CLOSED_COMPLETE_METADATA
METADATA_AND_LINEAR_PLAYOUT
METADATA_AND_RANDOM_ACCESS

This enumeration defines the opening modes available when calling **NewMxfFileInterface()**.

3.2 I_op1a_file Class Reference

```
#include <I_op1a_file.hpp>
```

This class should be used in order to create Op1a MXF file. Op1a MXF file contain a single **I_concrete_material** played as is.



Public Member Function

I_generic_material *get_current_generic_material ()
bool set_external_material (I_concrete_material *, const wchar_t*)
bool set_material (I_concrete_material *)

Constructor and Destructor Documentation

bool NewOp1aFileInterface (I_op1a_file **, const char *)

Retrieves a pointer on an **I_op1a_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

bool NewOpZeroFileInterface (I_op1a_file **, const char *)

Retrieves a pointer on an **I_op1a_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. OpZero files are a specialization of Op1a files containing a single video or audio track as well as partitioning properties adjusted to enable play while record. These files are usually recognized by Omneon servers. Note that the concrete material that will be added to the op1a (opzero) file should have been wrapped using the opzero wrapping mode. Return *false* if the allocation failed.

bool NewOp1aStreamInterface (I_op1a_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_op1a_file** interface. This function must be called when writing an Op1a file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Return *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool NewOpZeroStreamInterface (I_op1a_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_op1a_file** interface. This function must be called when writing an OpZero file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. OpZero files are a specialization of Op1a files containing a single video or audio track as well as partitioning properties adjusted to enable play while record. These files are usually recognized by Omneon servers. Note that the concrete material that will be added to the op1a (opzero) file should have been wrapped using the opzero wrapping mode. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool FreeOp1aFileInterface (I_op1a_file **)

Frees an **I_op1a_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

I_generic_material *get_current_generic_material ()

Returns the generic material that was created when calling **set_material()** or **set_external_material()**. The function will return NULL if none of these functions has been called.

bool set_external_material (I_concrete_material *, const wchar_t *)

Sets an audiovisual material originating from another MXF file to be externally referenced. When calling this function the referenced MXF file will be added to the list of external references for this Op1a file. The Op1a file will contain a copy of the concrete material but will not embed the audiovisual material contained in this material. The second parameter should hold the path toward the referenced MXF file. It can be relative or absolute. However, you should make sure that the path is correct relatively to the place where the Op1a file will be created. Returns *false* if an error occurred.

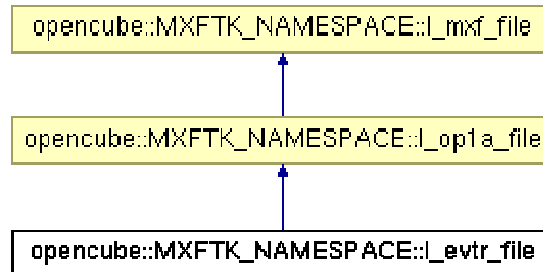
bool set_material (I_concrete_material *)

Sets an audiovisual material to embed in the Op1a file. This call also causes the automatic generation of the output material **I_generic_material** (similar to the **I_concrete_material** in the context of an Op1a operational pattern) that can be retrieved thanks to **output_material()**. Returns *false* if an error occurred.

3.3 I_evtr_file Class Reference

```
#include <I_evtr_file.hpp>
```

This class should be used in order to create Op1a MXF file matching the properties of those generated by SONY eVTR recording device. It is mandatory to provide IMX video and 8-channel AES audio input to the concrete material in order to generate MXF files compatible with SONY eVTR. If 8-channel AES audio is not available, it is also possible to provide WAVE audio and turn on the **WrapWaveAsAES8()** function.



Constructor and Destructor Documentation

bool NewEvtrFileInterface (I_evtr_file **, const char *)

Retrieves a pointer on an **I_evtr_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Note that the concrete material that will be added to the evtr file should have been wrapped using the evtr wrapping mode. Returns *false* if the allocation failed.

bool NewEvtrStreamInterface (I_evtr_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_evtr_file** interface. This function must be called when writing an eVTR file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Note that the concrete material that will be added to the evtr file should have been wrapped using the evtr wrapping mode. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

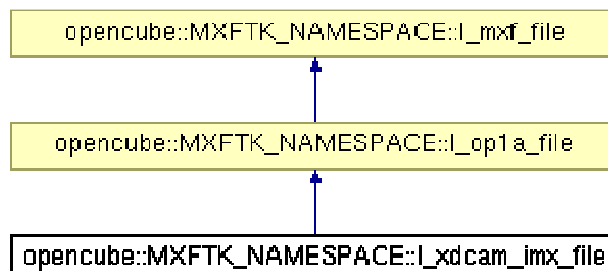
bool FreeEvtrFileInterface (I_evtr_file **)

Free an **I_evtr_file** interface. Return *false* if the deallocation failed.

3.4 I_xdcam_imx_file Class Reference

```
#include <I_xdcam_imx_file.hpp>
```

This class should be used in order to create Op1a MXF file matching the properties of those generated by SONY XDCam camcorder. It is mandatory to provide IMX video and 8-channel AES audio input to the concrete material in order to generate MXF files compatible with SONY XDCam. If 8-channel AES audio is not available, it is also possible to provide WAVE audio and turn on the **WrapWaveAsAES8()** function.



Constructor and Destructor Documentation

bool NewXdcamImxFileInterface (I_xdcam_imx_file **, const char *)

Retrieves a pointer on an **I_xdcam_imx_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Note that the concrete material that will be added to the evtr file should have been wrapped using the xdcam wrapping mode. Returns *false* if the allocation failed.

bool NewXdcamImxStreamInterface (I_xdcam_imx_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_xdcam_imx_file** interface. This function must be called when writing a D10 XDCam file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Note that the concrete material that will be added to the evtr file should have been wrapped using the xdcam wrapping mode. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

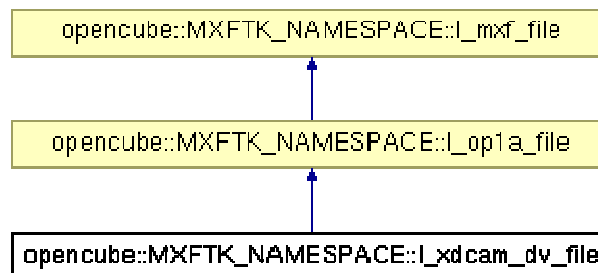
bool FreeXdcamImxFileInterface (I_xdcam_imx_file **)

Frees an **I_xdcam_imx_file** interface. Return *false* if the deallocation failed.

3.5 I_xdcam_dv_file Class Reference

```
#include <I_xdcam_dv_file.hpp>
```

This class should be used in order to create Op1a MXF file matching the properties of those generated by SONY XDCam camcorder. It is mandatory to provide DV IEC video and four AES audio inputs in order to generate these files. If AES audio is not available, it is also possible to provide mono WAVE audio and turn on the **WrapWaveAsAES()** function.



Constructor and Destructor Documentation

bool NewXdcamDvFileInterface (I_xdcam_dv_file **, const char *)

Retrieves a pointer on an **I_xdcam_dv_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Note that the concrete material that will be added to the evtr file should have been wrapped using the xdcam wrapping mode. Returns *false* if the allocation failed.

bool NewXdcamDvStreamInterface (I_xdcam_dv_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_xdcam_dv_file** interface. This function must be called when writing a DV XDCam file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Note that the concrete material that will be added to the evtr file should have been wrapped using the xdcam wrapping mode. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

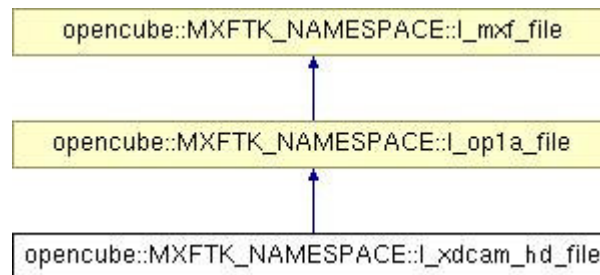
bool FreeXdcamDvFileInterface (I_xdcam_dv_file **)

Frees an **I_xdcam_dv_file** interface. Returns *false* if the deallocation failed.

3.6 I_xdcam_hd_file Class Reference

```
#include <I_xdcam_hd_file.hpp>
```

This class should be used in order to create Op1a MXF file matching the properties of those generated by SONY XDCam HD camcorder. It is mandatory to provide MPEG H-14 or MPEG MP@HL LongGOP video and four AES audio inputs in order to generate these files. If AES audio is not available, it is also possible to provide mono WAVE audio and turn on the **WrapWaveAsAES()** function.



Constructor and Destructor Documentation

bool NewXdcamHdFileInterface (I_xdcam_hd_file **, const char *)

Retrieves a pointer on an **I_xdcam_dv_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Note that the concrete material that will be added to the evtr file should have been wrapped using the xdcam wrapping mode. Returns *false* if the allocation failed.

bool NewXdcamHdStreamInterface (I_xdcam_hd_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_xdcam_hd_file** interface. This function must be called when writing a XDCam HD file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Note that the concrete material that will be added to the evtr file should have been wrapped using the xdcam wrapping mode. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

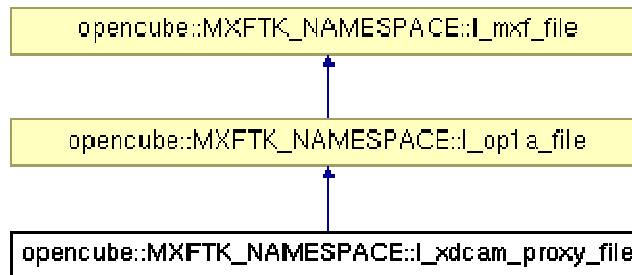
bool FreeXdcamHdFileInterface (I_xdcam_hd_file **)

Frees an **I_xdcam_hd_file** interface. Returns *false* if the deallocation failed.

3.7 I_xdcam_proxy_file Class Reference

```
#include <I_xdcam_proxy_file.hpp>
```

This class should be used in order to create Op1a MXF file matching the properties of those generated by SONY XDCam camcorder. It is mandatory to provide MPEG4 video and four 2-channel A-law audio inputs in order to generate these files. Standardization of the MPEG4 wrapping is not fully specified by the MXF norm; therefore you should be aware that some MXF decoders may state that XDCam proxy files contain MPEG2 video although they contain MPEG4 video.



Constructor and Destructor Documentation

bool NewXdcamProxyFileInterface (I_xdcam_proxy_file **, const char *)

Retrieves a pointer on an **I_xdcam_dv_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Note that the concrete material that will be added to the evtr file should have been wrapped using the xdcam wrapping mode. Returns *false* if the allocation failed.

bool NewXdcamProxyStreamInterface (I_xdcam_proxy_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_xdcam_dv_file** interface. This function must be called when writing a MPEG4 proxy XDCam file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Note that the concrete material that will be added to the evtr file should have been wrapped using the xdcam wrapping mode. Return *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

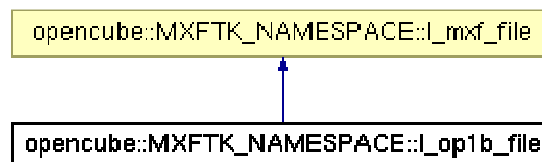
bool FreeXdcamDvFileInterface (I_xdcam_dv_file **)

Frees an **I_xdcam_dv_file** interface. Returns *false* if the deallocation failed.

3.8 I_op1b_file Class Reference

```
#include <I_op1b_file.hpp>
```

This class should be used in order to create Op1b MXF file. Op1b MXF files contain several **I_concrete_material** that will be played simultaneously. Omneon files with external references are usually built using an Op1b file referencing raw media files or OpZero MXF files.



Public Member Function

bool add_external_material (I_concrete_material *, const wchar_t *)

bool add_material (I_concrete_material *)

I_generic_material *get_current_generic_material ()

Constructor and Destructor Documentation

bool NewOp1bFileInterface (I_op1b_file **, const char *)

Retrieves a pointer on an **I_op1b_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

bool NewOp1bStreamInterface (I_op1b_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_op1b_file** interface. This function must be called when writing an Op1b file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool FreeOp1bFileInterface (I_op1b_file **)

Frees an **I_op1b_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

bool add_external_material (I_concrete_material *, const wchar_t *)

Adds an audiovisual material originating from another MXF file to be externally referenced. When calling this function the referenced MXF file will be added to the list of external references for this Op1b file. The Op1b file will contain a copy of the concrete material but will not embed the audiovisual material contained in this material. There is no limit to the number of **I_concrete_material** that can be added to an Op1b file. The current output material will be updated after each call to this function. The second parameter should hold the path toward the referenced MXF file. It can be relative or absolute. However, you should make sure that the path is correct relatively to the place where the Op1a file will be created. Returns *false* if an error occurred.

bool add_material (I_concrete_material *)

Adds an audiovisual material to embed in the Op1b file. There is no limit to the number of **I_concrete_material** that can be added to an Op1b file. This call also causes the automatic generation of the output material **I_generic_material** that can be retrieved thanks to **output_material()**. The output material will be updated after each call to this function. Returns *false* if an error occurred.

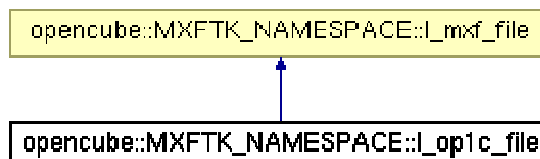
I_generic_material *get_current_generic_material ()

Returns the generic material that was created when calling **add_material()** or **add_external_material()**. The function will return NULL if none of these functions has been called.

3.9 I_op1c_file Class Reference

```
#include <I_op1b_file.hpp>
```

This class should be used in order to create Op1c MXF file. Op1c MXF files contain several output material (**I_generic_material**). Each of them is built as a collection of **I_concrete_material**'s tracks that will be played simultaneously. It can be seen as a collection of Op1b files.



Public Member Function

bool add_external_material_track (I_concrete_material *, I_concrete_track *, const wchar_t *)

bool add_material_track (I_concrete_material *, I_concrete_track *)

I_generic_material *get_current_generic_material ()

bool new_generic_material()

Constructor and Destructor Documentation

bool NewOp1cFileInterface (I_op1c_file **, const char *)

Retrieves a pointer on an **I_op1c_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

bool NewOp1cStreamInterface (I_op1c_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_op1c_file** interface. This function must be called when writing an Op1c file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool FreeOp1cFileInterface (I_op1c_file **)

Frees an **I_op1c_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

bool add_external_material_track (I_concrete_material *, I_concrete_track *, const wchar_t *)

Adds the concrete track of an audiovisual material originating from another MXF file to be externally referenced. When calling this function the referenced MXF file will be added to the list of external references for this Op1c file. The Op1c file will contain a copy of the concrete material but will not embed the audiovisual material contained in this material. The current output material will be updated after each call to this function. The third parameter should hold the path toward the referenced MXF file. It can be relative or absolute. However, you should make sure that the path is correct relatively to the place where the Op1c file will be created. Returns *false* if an error occurred.

bool add_material_track (I_concrete_material *, I_concrete_track *)

Adds the concrete track of an audiovisual material that will be embedded in the Op1c file. The current output material will be updated after each call to this function. Returns *false* if an error occurred.

bool new_generic_material (I_concrete_material *)

Creates a new generic material that will be added to the list of output material. All the following calls to **add_material_track()** and **add_external_material_track()** will add an output track to the newly created generic material. Returns *false* if an error occurred.

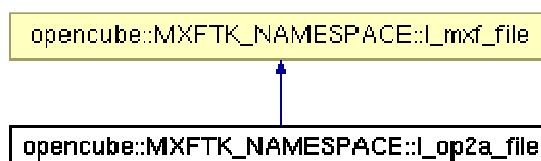
I_generic_material *get_current_generic_material ()

Returns the generic material that was created when calling **new_generic_material()**. You should always call this function after calling **add_material_track()** or **add_external_material_track()** in order to manipulate a valid material. You should never work on previous instances of a material returned by this function. The function will return NULL if there is no generic material currently defined in the file.

3.10 I_op2a_file Class Reference

```
#include <I_op2a_file.hpp>
```

This class should be used in order to create Op2a MXF files. Op2a MXF files contain several **I_concrete_material** that will be played sequentially.



Public Member Function

bool add_external_material (I_concrete_material *, const wchar_t *)
bool add_material (I_concrete_material *)
I_generic_material *get_current_generic_material ()
bool set_continuous_decoding (bool)

Constructor and Destructor Documentation

bool NewOp2aFileInterface (I_op2a_file **, const char *)

Retrieves a pointer on an **I_op2a_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

bool NewOp2aStreamInterface (I_op2a_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_op2a_file** interface. This function must be called when writing an Op2a file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool FreeOp2aFileInterface (I_op2a_file **)

Frees an **I_op2a_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

bool add_external_material (I_concrete_material *, const wchar_t *)

Adds an audiovisual originating from another MXF file to be externally referenced. When calling this function the referenced MXF file will be added to the list of external references for this Op2a file. The Op2a file will contain a copy of the concrete material but will not embed the audiovisual material contained in this material. There is no limit to the number of **I_concrete_material** that can be added to an Op2a file. The current output material will be updated after each call to this function. The second parameter should hold the path toward the referenced MXF file. It can be relative or absolute. However, you should make sure that the path is correct relatively to the place where the Op2a file will be created. Returns *false* if an error occurred.

bool add_material (I_concrete_material *)

Adds an audiovisual material to embed in the Op2a file. There is no limit to the number of **I_concrete_material** that can be added to an Op2a file. The current output material will be updated after each call to this function. Returns *false* if an error occurred.

I_generic_material *get_current_generic_material ()

Returns the generic material that was created when calling **add_material()** or **add_external_material()**. The function will return NULL if none of these functions has been called.

bool set_continuous_decoding (bool)

You should call this function to tell MXFTk if the concrete material added to the file can be continuously decoded. “Continuous decoding” means that no special processing is required at the junction of the concrete material being played. The following examples of files do not allow a continuous decoding:

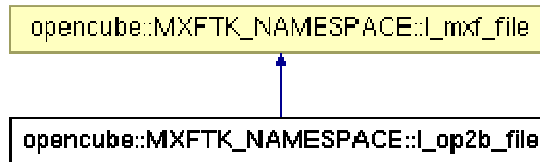
- A DV source followed by a MPEG source.
- A mono WAVE source followed by a stereo WAVE source.
- Two MPEG Long GOP sources cut so that a MPEG decoder will not be able to decode some of the frames.

Return *false* if an error occurred.

3.11 I_op2b_file Class Reference

```
#include <I_op2b_file.hpp>
```

This class should be used in order to create Op2b MXF files. Op2b MXF files contain several sets of **I_concrete_material** played simultaneously. The sets are then played sequentially. This is a combination of the operational pattern 1b and 2a.



Public Member Function

```

bool add_external_material(I_concrete_material *, const wchar_t *)
bool add_material (I_concrete_material *)
I_generic_material *get_current_generic_material ()
bool new_multiplexed_material ()
bool set_continuous_decoding (bool)

```

Constructor and Destructor Documentation

```
bool NewOp2bFileInterface (I_op2b_file **, const char *)
```

Retrieves a pointer on an **I_op2b_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

```
bool NewOp2bStreamInterface (I_op2b_file **, I_output_mxf_stream_task *)
```

Retrieves a pointer on an **I_op2b_file** interface. This function must be called when writing an Op2b file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

```
bool FreeOp2bFileInterface (I_op2b_file **)
```

Frees an **I_op2b_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

```
bool add_external_material (I_concrete_material *, const wchar_t *)
```

Adds an audiovisual material originating from another MXF file to be externally referenced. When calling this function the referenced MXF file will be added to the list of external references for this Op2b file. The Op2b file will contain a copy of the concrete material but will not embed the audiovisual material contained in this material. There is no limit to the number of **I_concrete_material** that can be added to an Op2b file. The current output material will be updated after each call to this function. The second parameter should hold the path toward the referenced MXF file. It can be relative or absolute. However, you should make sure that the path is correct relatively to the place where the Op1a file will be created. Returns *false* if an error occurred.

```
bool add_material (I_concrete_material *)
```

Adds an audiovisual material to embed in the Op2b file. It is added to the current set of multiplexed material. There is no limit to the number of **I_concrete_material** that can be added to an Op2b file. The current output material will be updated after each call to this function. Returns *false* if an error occurred.

bool new_muxed_material ()

Notify to MXFTk that the next concrete material added to this file will be in a new set of ganged material. For instance, in order to create the following Op2b file where A, B, C, D, E and F designate concrete material you should perform the following calls:

Output Material:

Track1: A followed by B followed by C

Track2: D followed by E followed by F

MXFTk calls:

new_muxed_material()

add_material(A)

add_material(D)

new_muxed_material()

add_material(B)

add_material(E)

new_muxed_material()

add_material(C)

add_material(F)

I_generic_material *get_current_generic_material ()

Returns the generic material that was created when calling **add_material()** or **add_external_material()**. The function will return NULL if none of these functions has been called.

bool set_continuous_decoding (bool)

You should call this function to tell MXFTk if the concrete material added to the file can be continuously decoded. “Continuous decoding” means that no special processing is required at the junction of the concrete material being played. The following examples of files do not allow a continuous decoding:

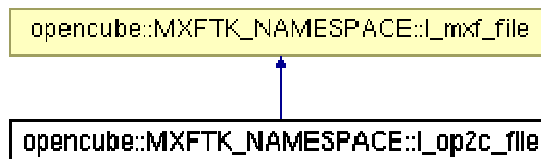
- A DV source followed by a MPEG source.
- A mono WAVE source followed by a stereo WAVE source.
- Two MPEG Long GOP sources cut so that a MPEG decoder will not be able to decode some of the frames.

Returns *false* if an error occurred.

3.12 I_op2c_file Class Reference

```
#include <I_op2c_file.hpp>
```

This class should be used in order to create Op2c MXF files. Op2c MXF files contain several output material (**I_generic_material**). Each of them contains several sets of **I_concrete_material** played simultaneously. The sets are then played sequentially. It can be seen as a collection of Op2b files.



Public Member Function

bool add_external_material_track(I_concrete_material *, I_concrete_track *, const wchar_t *)

bool add_material_track (I_concrete_material *, I_concrete_track *)

I_generic_material *get_current_generic_material ()

bool new_generic_material ()

bool new_muxed_track ()

bool set_continuous_decoding (bool)

Constructor and Destructor Documentation

bool NewOp2cFileInterface (I_op2c_file **, const char *)

Retrieves a pointer on an **I_op2c_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

bool NewOp2cStreamInterface (I_op2c_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_op2c_file** interface. This function must be called when writing an Op2c file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool FreeOp2cFileInterface (I_op2c_file **)

Frees an **I_op2c_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

bool add_external_material_track (I_concrete_material *, I_concrete_track *, const wchar_t *)

Adds the concrete track of an audiovisual material originating from another MXF file to be externally referenced. When calling this function the referenced MXF file will be added to the list of external references for this Op2c file. The Op2c file will contain a copy of the concrete material but will not embed the audiovisual material contained in this material. The current output material will be updated after each call to this function. The third parameter should hold the path toward the referenced MXF file. It can be relative or absolute. However, you should make sure that the path is correct relatively to the place where the Op2c file will be created. Returns *false* if an error occurred.

bool add_material_track (I_concrete_material *, I_concrete_track *)

Adds the concrete track of an audiovisual material that will be embedded in the Op2c file. The current output material will be updated after each call to this function. Returns *false* if an error occurred.

bool new_generic_material ()

Creates a new generic material that will be added to the list of output material. All the following calls to **add_material_track()** and **add_external_material_track()** will add an output track to the newly created generic material. Returns *false* if an error occurred.

I_generic_material *get_current_generic_material ()

Returns the generic material that was created when calling **new_generic_material()**. You should always call this function after calling **add_material_track()** or **add_external_material_track()** in order to manipulate a valid material. You should never work on previous instances of a material returned by this function. The function will return NULL if there is no generic material currently defined in the file.

bool new_multiplexed_track ()

Notifies MXFTk that the next concrete material added to this file will be in a new set of ganged material. For instance, in order to create the following Op2c generic material where A, B, C, D, E and F designate concrete tracks you should perform the following calls:

Output Material:

Track1: A followed by B followed by C

Track2: D followed by E followed by F

MXFTk calls:

new_generic_material()

new_multiplexed_track()

add_material_track(A)

add_material_track(D)

new_multiplexed_track()

add_material_track(B)

add_material_track(E)


```
new_multiplexed_track()
add_material_track(C)
add_material_track(F)
```

The process is repeated for each output material to be created.

bool set_continuous_decoding (bool)

You should call this function to tell MXFTk if the concrete material added to the file can be continuously decoded. “Continuous decoding” means that no special processing is required at the junction of the concrete material being played. The following examples of files do not allow a continuous decoding:

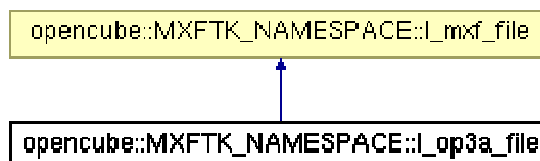
- A DV source followed by a MPEG source.
- A mono WAVE source followed by a stereo WAVE source.
- Two MPEG Long GOP sources cut so that a MPEG decoder will not be able to decode some of the frames.

Returns *false* if an error occurred.

3.13 I_op3a_file Class Reference

```
#include <I_op3a_file.hpp>
```

This class should be used in order to create Op3a MXF files. Op3a MXF files contain several tracks cut from various **I_concrete_material**’s tracks that will be played sequentially.



Public Member Function

```
bool add_external_material (I_concrete_material *, I_timecode *, I_timecode *, const wchar_t *)
bool add_material (I_concrete_material *, I_timecode *, I_timecode *)
I_generic_material *get_current_generic_material ()
bool set_continuous_decoding (bool)
```

Constructor and Destructor Documentation

bool NewOp3aFileInterface (I_op3a_file **, const char *)

Retrieves a pointer on an **I_op3a_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

bool NewOp3aStreamInterface (I_op3a_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_op3a_file** interface. This function must be called when writing an Op3a file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool FreeOp3aFileInterface (I_op3a_file **)

Frees an **I_op3a_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

bool add_external_material (I_concrete_material *, I_timecode *, I_timecode *, const wchar_t *)

Adds an audiovisual originating from another MXF file to be externally referenced. When calling this function the referenced MXF file will be added to the list of external references for this Op3a file. The Op3a file will contain a copy of the concrete material but will not embed the audiovisual material contained in this material. There is no limit to the number of **I_concrete_material** that can be added to an Op3a file. The current output material will be updated after each call to this function. The second and third parameters define the portion of the concrete material's tracks that will be appended to the generic material. The time code values should be expressed relatively to the time code track of the concrete material. All the tracks from the concrete material will be considered. The fourth parameter should hold the path toward the referenced MXF file. It can be relative or absolute. However, you should make sure that the path is correct relatively to the place where the Op3a file will be created. Returns *false* if an error occurred.

bool add_material (I_concrete_material *, I_timecode *, I_timecode *)

Adds an audiovisual material to embed in the Op2a file. There is no limit to the number of **I_concrete_material** that can be added to an Op3a file. The current output material will be updated after each call to this function. The second and third parameters define the portion of the concrete material's tracks that will be appended to the generic material. The time code values should be expressed relatively to the time code track of the concrete material. All the tracks from the concrete material will be considered. Returns *false* if an error occurred.

I_generic_material *get_current_generic_material ()

Returns the generic material that was created when calling **add_material()** or **add_external_material()**. The function will return NULL if none of these functions has been called.

bool set_continuous_decoding (bool)

You should call this function to tell MXFTk if the concrete material added to the file can be continuously decoded. "Continuous decoding" means that no special processing is required at the junction of the concrete material being played. The following examples of files do not allow a continuous decoding:

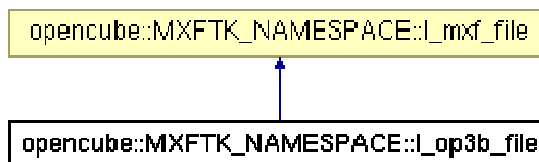
- A DV source followed by a MPEG source.
- A mono WAVE source followed by a stereo WAVE source.
- Two MPEG Long GOP sources cut so that a MPEG decoder will not be able to decode some of the frames.

Returns *false* if an error occurred.

3.14 I_op3b_file Class Reference

```
#include <I_op3b_file.hpp>
```

This class should be used in order to create Op3b MXF files. Op3b MXF files contain several sets of tracks (cut from various **I_concrete_material**'s tracks) played simultaneously. The sets are then played sequentially.



Public Member Function

bool add_external_material_track (I_concrete_material *, I_concrete_track *, I_timecode *, I_timecode *, const wchar_t *)

bool add_material_track (I_concrete_material *, I_concrete_track *, I_timecode *, I_timecode *)

I_generic_material *get_current_generic_material ()

bool new_generic_material_track ()

bool set_continuous_decoding (bool)

Constructor and Destructor Documentation

bool NewOp3bFileInterface (I_op3b_file **, const char *)

Retrieves a pointer on an **I_op3b_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

bool NewOp3bStreamInterface (I_op3b_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_op3b_file** interface. This function must be called when writing an Op3b file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool FreeOp3bFileInterface (I_op3b_file **)

Frees an **I_op3b_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

bool add_external_material_track (I_concrete_material *, I_concrete_track *, I_timecode *, I_timecode *, const wchar_t *)

Adds the concrete track of an audiovisual material originating from another MXF file to be externally referenced. When calling this function the referenced MXF file will be added to the list of external references for this Op3b file. The Op3b file will contain a copy of the concrete material but will not embed the audiovisual material contained in this material. There is no limit to the number of **I_concrete_material** that can be added to an Op3b file. The current output material will be updated after each call to this function. The third and fourth parameters define the portion of the concrete material’s track that will be appended to the generic material. The time code values should be expressed relatively to the time code track of the concrete material. The fifth parameter should hold the path toward the referenced MXF file. It can be relative or absolute. However, you should make sure that the path is correct relatively to the place where the Op1a file will be created. Returns *false* if an error occurred.

bool add_material_track (I_concrete_material *, I_concrete_track *, I_timecode *, I_timecode *)

Adds the concrete track of an audiovisual material to embed in the Op3b file. It is added to the current set of multiplexed material. There is no limit to the number of **I_concrete_material** that can be added to an Op3b file. The current output material will be updated after each call to this function. The third and fourth parameters define the portion of the concrete material’s track that will be appended to the generic material. The time code values should be expressed relatively to the time code track of the concrete material. Returns *false* if an error occurred.

I_generic_material *get_current_generic_material ()

Returns the generic material that was created when calling **add_material_track()** or **add_external_material_track()**. The function will return NULL if none of these functions has been called.

bool new_generic_material_track ()

Notifies MXFTk that a new track should be created for the current generic material. The next portions of concrete material’s tracks will be added to this new track. For instance, in order to create the following Op3b file where A, B, C, D, E and F designate concrete material tracks you should perform the following calls:

Output Material:

Track1: A[tc_inA, tc_outA] followed by B[tc_inB, tc_outB] followed by C[tc_inC, tc_outC]

Track2: D[tc_inD, tc_outD] followed by E[tc_inE, tc_outE] followed by F[tc_inF, tc_outF]

MXFTk calls:

```
new_generic_material_track()
```

```
add_material_track(A, tcin_A, tcout_A)
```

```
add_material_track(B, tcin_B, tcout_B)
```

```
add_material_track(C, tcin_C, tcout_C)
```

```
new_generic_material_track()
```



```
add_material_track(D, tcin_D, tcout_D)
add_material_track(E, tcin_E, tcout_E)
add_material_track(F, tcin_F, tcout_F)
```

bool set_continuous_decoding (bool)

You should call this function to tell MXFTk if the concrete material added to the file can be continuously decoded. “Continuous decoding” means that no special processing is required at the junction of the concrete material being played. The following examples of files do not allow a continuous decoding:

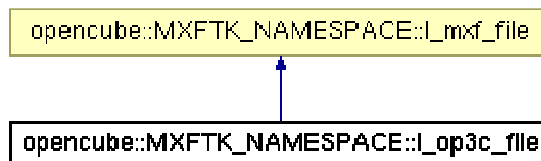
- A DV source followed by a MPEG source.
- A mono WAVE source followed by a stereo WAVE source.
- Two MPEG Long GOP sources cut so that a MPEG decoder will not be able to decode some of the frames.

Returns *false* if an error occurred.

3.15 I_op3c_file Class Reference

```
#include <I_op3c_file.hpp>
```

This class should be used in order to create Op3c MXF files. Op3c MXF files contain several output material (**I_generic_material**). Each of them contains several sets of tracks (cut from **I_concrete_material**’s tracks) played simultaneously. The sets are then played sequentially. It can be seen as a collection of Op3b files.



Public Member Function

```
bool add_external_material_track(I_concrete_material *, I_concrete_track *, I_timecode *,
    I_timecode *, const wchar_t *)
bool add_material_track (I_concrete_material *, I_concrete_track *, I_timecode *, I_timecode *)
I_generic_material *get_current_generic_material ()
bool new_generic_material ()
bool new_generic_material_track ()
bool set_continuous_decoding (bool)
```

Constructor and Destructor Documentation

bool NewOp3cFileInterface (I_op3c_file **, const char *)

Retrieves a pointer on an **I_op3c_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

bool NewOp3cStreamInterface (I_op3c_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_op3c_file** interface. This function must be called when writing an Op3c file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool FreeOp3cFileInterface (I_op3c_file **)

Frees an **I_op3c_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

bool add_external_material_track (I_concrete_material *, I_concrete_track *, I_timecode *, I_timecode *, const wchar_t *)

Adds the concrete track of an audiovisual material originating from another MXF file to be externally referenced. When calling this function the referenced MXF file will be added to the list of external references for this Op3c file. The Op3c file will contain a copy of the concrete material but will not embed the audiovisual material contained in this material. The current output material will be updated after each call to this function. The third and fourth parameters define the portion of the concrete material's track that will be appended to the generic material. The time code values should be expressed relatively to the time code track of the concrete material. The fifth parameter should hold the path toward the referenced MXF file. It can be relative or absolute. However, you should make sure that the path is correct relatively to the place where the Op2c file will be created. Returns *false* if an error occurred.

bool add_material_track (I_concrete_material *, I_concrete_track *, I_timecode *, I_timecode *)

Adds the concrete track of an audiovisual material that will be embedded in the Op2c file. The current output material will be updated after each call to this function. The third and fourth parameters define the portion of the concrete material's track that will be appended to the generic material. The time code values should be expressed relatively to the time code track of the concrete material. Returns *false* if an error occurred.

I_generic_material *get_current_generic_material ()

Returns the generic material that was created when calling **new_generic_material()**. You should always call this function after calling **add_material_track()** or **add_external_material_track()** in order to manipulate a valid material. You should never work on previous instances of a material returned by this function. The function will return NULL if there is no generic material currently defined in the file.

bool new_generic_material ()

Creates a new generic material that will be added to the list of output material. All the following calls to **new_generic_material_track()** will add an output track to the newly created generic material. Returns *false* if an error occurred.

bool new_generic_material_track ()

Notifies MXFTk that a new track should be created for the current generic material. The next portions of concrete material's tracks will be added to this new track. For instance, in order to create the following Op3b file where A, B, C, D, E and F designate concrete material tracks you should perform the following calls:

Output Material:

Track1: A[tc_inA, tc_outA] followed by B[tc_inB, tc_outB] followed by C[tc_inC, tc_outC]

Track2: D[tc_inD, tc_outD] followed by E[tc_inE, tc_outE] followed by F[tc_inF, tc_outF]

MXFTk calls:

new_generic_material()

new_generic_material_track()

add_material_track(A, tcin_A, tcout_A)

add_material_track(B, tcin_B, tcout_B)

add_material_track(C, tcin_C, tcout_C)

new_generic_material_track()

add_material_track(D, tcin_D, tcout_D)

add_material_track(E, tcin_E, tcout_E)

add_material_track(F, tcin_F, tcout_F)

The process is repeated for each output material to be created.

bool set_continuous_decoding (bool)

You should call this function to tell MXFTk if the concrete material added to the file can be continuously decoded. "Continuous decoding" means that no special processing is required at the junction of the concrete material being played. The following examples of files do not allow a continuous decoding:

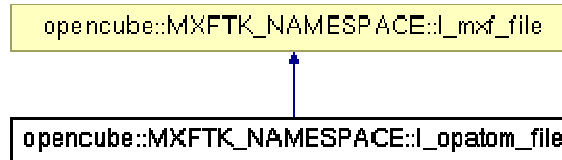
- A DV source followed by a MPEG source.
- A mono WAVE source followed by a stereo WAVE source.

- Two MPEG Long GOP sources cut so that a MPEG decoder will not be able to decode some of the frames. Return *false* if an error occurred.

3.16 I_opatom_file Class Reference

```
#include <I_opatom_file.hpp>
```

This class should be used in order to create OpAtom MXF files.



Public Member Function

```
I_umid* get_concrete_material_umid () const  
bool set_material (I_concrete_material *)
```

Constructor and Destructor Documentation

```
bool NewOpAtomFileInterface (I_opatom_file **, const char *)
```

Retrieves a pointer on an **I_opatom_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

```
bool NewOpAtomStreamInterface (I_opatom_file **, I_output_mxf_stream_task *)
```

Retrieves a pointer on an **I_opatom_file** interface. This function must be called when writing an OpAtom file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

```
bool NewDcpOpAtomFileInterface (I_opatom_file **, const char *)
```

Retrieves a pointer on an **I_opatom_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Return *false* if the allocation failed
Builds SMPTE429 OpAtom files for Digital Cinema Package(DCP).

```
bool NewDcpOpAtomStreamInterface (I_opatom_file **, I_output_mxf_stream_task *)
```

Retrieves a pointer on an **I_opatom_file** interface. This function must be called when writing an OpAtom file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Return *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.
Builds SMPTE429 OpAtom files for Digital Cinema Package(DCP).

```
bool FreeOpAtomFileInterface (I_opatom_file **)
```

Frees an **I_opatom_file** interface. Returns *false* if the deallocation failed.

Member Function Documentation

```
I_umid* get_concrete_material_umid () const
```


Returns the Unique Material Identifier of the audiovisual material embedded in this OpAtom file. This is the same UMID as the one returned by **I_concrete_material::get_umid()**. This UMID is used to link OpAtom files meant to be played together. Therefore, the function **get_umid()** is repeated here for the user's convenience.

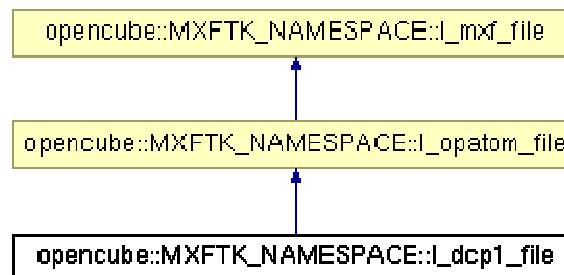
bool set_material (I_concrete_material *)

Sets an audiovisual material to embed in the OpAtom file. This call also causes the automatic generation of the output material **I_generic_material** that can be retrieved thanks to **output_material()**. Due to the nature of OpAtom files, concrete material containing more than one audiovisual track will not be accepted. Returns *false* if an error occurred.

3.17 I_dcp1_file Class Reference

```
#include <I_dcp1_file.hpp>
```

This class should be used in order to create OpAtom Digital Cinema Package MXF files compatible with Doremi and Dolby D-Cinema servers.



Constructor and Destructor Documentation

bool NewDcp1FileInterface (I_dcp1_file **, const char *)

Retrieves a pointer on an **I_dcp1_file** interface. The second parameter designates the complete path toward the MXF file to be written upon calling of the **flush()** function. Returns *false* if the allocation failed.

bool NewDcp1StreamInterface (I_dcp1_file **, I_output_mxf_stream_task *)

Retrieves a pointer on an **I_dcp1_file** interface. This function must be called when writing an OpAtom file on a linear streaming device (e.g. a videotape recorder, an IEEE1394 port, etc.). The **I_output_mxf_stream_task** is a class derived by MXFTk user to provide its own implementation to feed the streaming device while output data is built by MXFTk. Returns *false* if the allocation failed. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool FreeDcp1FileInterface (I_dcp1_file **)

Frees an **I_dcp1_file** interface. Returns *false* if the deallocation failed.

3.18 I_opatom_assembler Class Reference

```
#include <I_opatom_file.hpp>
```

This class lets you manipulate OpAtom files altogether. While creating OpAtom files it enables to build complex editing by parallelizing or serializing OpAtom files. Linked OpAtom files share the same **I_generic_material** that references **I_concrete_material** from several **I_opatom_file**.

Public Member Functions

```
bool parallelize (I_opatom_file *, I_opatom_file *)
bool serialize (I_opatom_file *, I_opatom_file *)
bool synchronize (mxf_file_list *)
```

Constructor and Destructor Documentation

```
bool NewOpAtomAssemblerInterface (I_opatom_assembler **)
    Retrieves a pointer on an I_opatom_assembler interface. Returns false if the allocation failed.

bool FreeOpAtomAssemblerInterface (I_opatom_assembler **)
    Frees an I_opatom_assembler interface. Returns false if the deallocation failed.
```

Member Functions Documentation

```
bool parallelize (I_opatom_file *, I_opatom_file *)
    This function is used when creating OpAtom files. It links I_opatom files so that they will be played together when they will be read back. If we define // as the symbol for calling the parallelize() function, then file1//file2 and then file2//file3 will cause the three OpAtom files to be played together. Once two OpAtom files are parallelized they share the same I_generic_material and represent the same editing. Similarly, doing file1//file2 and file3//file4 and then finally file1//file3 will cause the four files to be played simultaneously and share the same I_generic_material. The function will return false if an error occurred.
```

```
bool serialize (I_opatom_file *, I_opatom_file *)
    This function is used when creating OpAtom files. It links I_opatom files so that they will be played one after the other. If we define + as the symbol for calling the serialize() function, then file1+file2 and then file2+file3 will cause the three OpAtom files to be played in a sequence file1->file2->file3. Once two OpAtom files are serialized they share the same I_generic_material and represent the same editing. Similarly, doing file1+file2 and file1+file3 and then finally file1+file4 will cause the four files to be played in a sequence and share the same I_generic_material. When serializing files you must ensure that their I_generic_material share the same track structure (i.e. you can not serialize a video-only OpAtom file with a sound-only OpAtom file). The generic material of each OpAtom files being serialized must contain exactly the same number of video track, audio track, data track and metadata track. The function will return false if an error occurred.
```

*A note about **parallelize()** and **serialize()**:*

We provide here an example of editing with OpAtom files. You may also refer to the example “opatom_wrapper” from your installation directory.

Performing
 Audio_file1//Video_file2 then,
 Audio_file3//Video_file4 then,
 Audio_file1+Audio_file3 will produce an **I_generic_material** with two tracks:

Audio Track: embedded audio from file1 followed by embedded audio from file2.
 Video Track: embedded video from file2 followed by embedded video from file4.

It corresponds to the editing of an operational pattern 2b. Therefore the four OpAtom files are OpAtom_2b files sharing the same generic material.

Furthermore performing
 Audio_file1+Audio_file3 then,
 Video_file2+Video_file4 then,
 Audio_file1//Video_file2 would have produced exactly the same result.

bool synchronize (mxf_file_list *)

This function is used when reading OpAtom files. The user should provide a list of **I_opatom_file** that share the same editing. The call to this function will ensure the correct linkage between **I_concrete_material** of each OpAtom file. It enables the **I_generic_material** of each file to access seamlessly all the **I_concrete_material** as if they were in the same file. For instance let's assume we have two OpAtom_1b files and that the **I_generic_material** from file1 uses the video track stored in the **I_concrete_material** of file2. Before synchronization, the file1 has no reference on the file2 and therefore its **I_generic_material** can not playback the video. However after synchronizing both files, the same **I_generic_material** from file1 will become able to playback the video. You may refer to the example "opatom_unwrapper" that provides a sample code for synchronizing any editing of OpAtom files. Trying to synchronize files that are not related will cause an error.

3.19 Panasonic P2 Functions Reference

```
#include <I_opatom_file.hpp>
```

bool BuildP2XML (const char*)

This function builds the P2 XML file stored in the Clip directory from a P2 video file. The first parameter should contain the complete path to this P2 file and it should be in its P2 directory structure (Contents/Video) and all its associated audio files must be in their directory (Contents/Audio). This function is intended to be used in order to rebuild the XML clip after performing a partial restore on a set of P2 files. The function returns *false* if an error occurred.

bool BuildP2XMLBuffer (I_mxf_file*, const char*, size_t*, unsigned char*)

This function builds the P2 XML file usually stored in the Clip directory from a P2 video file. The first parameter should reference a P2 file already linked to its associated P2 files thanks to `opatom_assembler`. The second parameter contains the base name of this set of P2 files. Finally the XML will be written in the unsigned char* buffer and its size will be set in the size_t* parameter. This function is similar to **BuildP2XML()** however it is convenient when you do not have access to the on-disk P2 files. Returns *false* if an error occurred.

bool CancelNewP2Shot ()

Cancels the P2 creation thread currently running. This call is asynchronous so the thread is not necessarily terminated when exiting this function. Returns *false* if an error occurred.

bool NewP2Shot (locators*, const char*, const char*, I_timecode*, I_umid*)

This function should be used in order to create OpAtom MXF files matching the properties of those generated by Panasonic P2 camcorder. The first parameter is the list of path to the source files to be wrapped (this should be a DVCPRO 25 or 50 along with two or four 2-channel AES/WAVE audio inputs). The second parameter is the path to the output directory. The third parameter is the base name for the files being generated and it must be strictly 6 characters long. Finally the two last optional parameters let you specify the starting timecode and the UMID of the sequence being generated. If you do not set these parameters, MXFTk will generate default ones for you.

This function will build a directory containing three or five Opatom files as well as an XML file. It corresponds to the structure found on P2 cards.

It will launch a thread so you need to call the function **WaitEndNewP2Shot()** to make sure the creation process is completed.

Returns *false* if an error occurred.

bool NewP2ShotFromStream (I_essence_stream_task*, I_essence_stream_task*, I_essence_stream_task*, I_essence_stream_task*, const char*, const char*, I_timecode*, I_umid*)

This function should be used in order to create OpAtom MXF files matching the properties of those generated by Panasonic P2 camcorder. This function should be called when working in a streaming environment. The first parameter is the streaming interface that will be used to manipulate the DVCPRO 25 or 50 stream. The four following `I_essence_stream_task` will be used to define the AES/WAVE streams. The second parameter is the path to the output directory. The third parameter is the base name for the files being generated and it must be strictly 6

characters long. Finally the two last optional parameters let you specify the starting timecode and the UMID of the sequence being generated. If you do not set these parameters, MXFTk will generate default ones for you. This function will build a directory containing three or five Opatom files as well as an XML file. This corresponds to the structure found on P2 cards.

It will launch a thread so you need to call the function **WaitEndNewP2Shot()** to make sure the creation process is completed.

Returns *false* if an error occurred.

double ProgressNewP2Shot ()

Returns the current progress of the P2 creation thread. The returned value is comprised between 0.0 (beginning) and 1.0 (end). You should not rely on this function to know if the thread has terminated.

bool WaitEndNewP2Shot ()

This function waits for the completion of the process launched by **NewP2Shot()** or **NewP2ShotFromStream()**. You should always call this function to ensure thread termination.

The function returns *false* if an error occurred.

4. Material

Two families of materials can be distinguished. On one side, we can find the concrete materials and on the other one the virtual materials. Concrete materials grant access to the audiovisual data (a clip in a given format) whereas virtual materials provide tools to manage the edition of the concrete materials, as well as their metadata and timecode.

TAB. 3: Materials

concrete	Virtual
<code>I_concrete_material</code>	<code>I_metadata_material</code>
	<code>I_timecode_material</code>
	<code>I_generic_material</code>

As seen in the preliminaries of this user guide, a material contains a given number of tracks. These tracks are used to spot the nature of the data held by this material (video, audio, data or metadata).

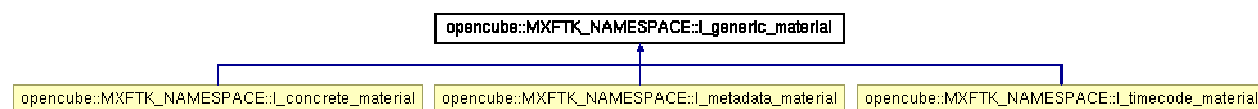
In the following sections, we will first go through the classes `I_generic_material` and `I_track` (representing editing information), then we will examine the concrete material classes `I_concrete_material` and `I_concrete_track` (representing audiovisual material) and finally we will conclude with the metadata material `I_metadata_material` and timecode specification `I_timecode_material`.

“Edit unit” is mentioned several times in this user guide. According to the MXF terminology, an “edit unit” is the indivisible unit of an MXF track (generally set to a frame or a field on a video track). It is the smallest addressable data unit of a track.

4.1 `I_generic_material` Class Reference

```
#include <I_generic_material.hpp>
```

This class describes an edition of the embedded material. Therefore, such a material is the play out of the `mxfile`. It contains references to audiovisual sources but does not contain essence data or tools to access to the binary data. Generic materials have a UMID (SMPTE Unique Material Identifier, see [330M]). MXFTk user can set its value after creating the material or leave the default one automatically set by the API.



Public Member Functions

```

bool delete_metadata_track (I_track *)
bool free_tracks (track_list *)
const wchar_t *get_material_name (size_t &) const
I_timecode_material *get_timecode () const
I_umid *get_umid () const
track_list *metadata_tracks ()
I_track *new_metadata_track (track_type, int64_t, rational *)
I_track *new_stream_metadata_track (I_input_metadata_stream_task *, track_type, int64_t, rational *)
bool set_material_name (const wchar_t *, size_t)
bool set_timecode (I_timecode_material *)
bool set_umid (const I_umid *)
track_list *source_tracks ()
const char *type () const
  
```


Member Functions Documentation

bool delete_metadata_track (I_track *)

Deletes a metadata track and all its associated metadata trees. Returns *false* if an error occurred.

bool free_tracks (track_list *)

Frees the list returned by `metadata_tracks()` or `source_tracks()`. Returns *false* if an error occurred.

const wchar_t *get_material_name (size_t &) const

Gets the Unicode UTF16 name of the current material. The *size_t* will contain the length of the returned wide string (in bytes). Returned buffer must not be deleted. Please note that on the Linux platform, the size of a *wchar_t* is 4 bytes. Hence, only 2 bytes of the *wchar_t* are effectively used when reading UTF16 strings. Returns NULL if the material is unnamed.

I_timecode_material *get_timecode () const

Retrieves an **I_timecode_material** interface defining the play out timecode of this material. Returned pointer must not be deleted.

I_umid *get_umid () const

Returns the value of the Unique Material Identifier uniquely identifying the current material. The purpose of this UMID is explained in the **I_umid** class reference. Returned pointer must be deleted using the function `FreeUmidInterface()`.

track_list *metadata_tracks ()

Gets the list of descriptive metadata tracks from this material. Each track can be a *timeline_metadata*, an *event_metadata* or a *static_metadata* track. Returned tracks can be edited but must not be destroyed. The list can be freed at any time using `free_tracks()`.

I_track *new_metadata_track (track_type, int64_t, rational *)

Creates a new metadata track to be filled with **I_metadata_material**. This track can be set to a *timeline_metadata*, an *event_metadata* or a *static_metadata* track. Following parameters designate the origin of the track measured in edit units as well as its edit rate (number of edit units in a second – the edit unit frequency). The **rational** value can be destroyed right after calling this function.

I_track *new_stream_metadata_track (I_input_metadata_stream_task *, track_type, int64_t, rational*)

Creates a new metadata track to be filled with **I_metadata_material**. This function can be used when you need to record metadata on-the-fly while creating an MXF file (the metadata you want to write is sent from a streaming device). The **I_input_metadata_stream_task** is a class derived by MXFTk user to provide its own implementation to feed MXFTk with metadata as it is received from the streaming device. The *track_type* can be set to a *timeline_metadata*, an *event_metadata* or a *static_metadata* track. Following parameters designate the origin of the track measured in edit units as well as its edit rate (number of edit units in a second – the edit unit frequency). The **rational** value can be destroyed right after calling this function. Please refer to the “Streaming” chapter for a complete overview of the streaming capabilities of MXFTk.

bool set_material_name (const wchar_t *, size_t)

Sets the name of this material. The first parameter contains the string in Unicode UTF16 and the second one the string's length in bytes. The *wchar_t** pointer can be destroyed after calling this function.

bool set_timecode (I_timecode_material *)

Sets a new timecode material. The previous one (if any) will be automatically destroyed. Within an **I_generic_material**, the timecode material must define a continuous timecode (in other words there should not be several timecode redefinitions within the timecode material) as the output of an MXF file is meant to be linear. Nonetheless, the timecode material of an **I_concrete_material** can be discontinuous. The new **I_timecode_material** will be freed upon destruction of the current material.

bool set_umid (const I_umid *)

Sets a new Unique Material Identifier for the current material. Older one will be automatically destroyed. A copy of the UMID is performed upon calling of this function so that the user remains responsible for the deletion of the **I_umid** pointer.

track_list *source_tracks ()

Returns a list of tracks representing the audiovisual content. Within an **I_generic_material** these tracks define the file play out. In an **I_concrete_material**, the **I_track** should be directly cast into **I_concrete_track***. Hence, these tracks will hold the embedded audiovisual binary data. Items of the list must not be destroyed. The list can be freed at any time using **free_tracks()**.

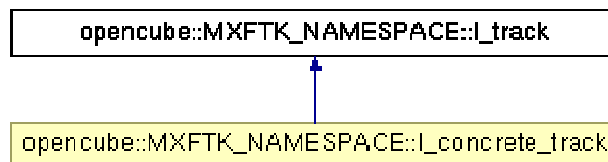
const char *type () const

Returns a string categorizing the current material (“Generic Material “, “Concrete Material “, “Metadata Material” or “Timecode Material”). Returned pointer must not be destroyed.

4.2 I_track Class Reference

```
#include <I_track.hpp>
```

This class represents a generic material track. Each material is likely to contain several tracks. An **I_track** is used to store metadata or editing information. **I_concrete_track** embed the audiovisual data while **I_track** reference **I_concrete_track** to build up a complete editing.



Public Member Functions

```

bool add_metadata (I_metadata_material *, track_list *)
int64_t duration ()
rational *edit_rate ()
I_timecode *edit_unit_to_timecode (int64_t)
ordered_track_item_list *elements ()
bool free_elements (ordered_track_item_list *) const
I_timecode *get_end_timecode ()
I_timecode *get_start_timecode ()
uint32_t get_track_id () const
const wchar_t *get_track_name (size_t &) const
ordered_track_item_list *item_seek_eu (int64_t)
ordered_track_item_list *item_seek_timecode (const I_timecode *)
int64_t origin ()
bool remove_metadata (I_metadata_material *)
bool set_track_name (const wchar_t *, size_t)
int64_t timecode_to_edit_unit (const I_timecode *)
track_type type () const
const char *type_name () const
  
```

Member Functions Documentation

bool add_metadata (I_metadata_material *, track_list *)

This function will be operative only on a metadata track; no metadata material can be appended to *picture*, *sound* or

data tracks. The **I_metadata_material** will be added to the track according to the following rules:

-*timeline_metadata*: metadata is appended right after the last metadata material already on the track. It is added in a time-linear fashion: metadata materials are continuously appended one after the other. The duration property of the metadata to be added is set in the **I_metadata_material** containing it. Therefore duration of the track is the sum of the **I_metadata_material** duration.

-*event_metadata*: metadata is added at a position set by the **I_metadata_material** containing it. This material also defines the duration of this metadata

-*static_metadata*: the **I_metadata_material** documents globally the **I_generic_material** containing this metadata track. Therefore, it does not contain any time or duration references.

The second parameter specifies a list of tracks that this metadata annotates. Thus, the list must contain pointers on **I_track/I_concrete_track** from the material containing this metadata track.

Adding metadata thanks to this function creates an **I_dm_segment** on the track. Return *false* if an error occurred.

int64_t duration ()

Returns the duration of the track measured in edit units. A negative value means that the duration is not known yet.

rational *edit_rate ()

Returns the edit rate of this track (number of edit units per second). Returned pointer should be destroyed using **FreeRationalInterface()**.

I_timecode *edit_unit_to_timecode (int64_t)

Converts the edit unit into a timecode value on this track. Returns NULL if the timecode value cannot be computed. Returned timecode value should be freed using **FreeTimecodeInterface()**.

ordered_track_item_list *elements ()

Returns a time-linearly ordered list of items from this track. Each **I_track_item** returned by this function can be either cast into an **I_source_clip**, **I_dm_source_clip** or **I_dm_segment** (respectively thanks to **source_clip_cast()**, **dm_source_clip_cast()** or **dm_segment_cast()**). Track items are the indivisible segments of data embedded on a track: several track items butted to each other (or overlapping) materialize a track. The list can be freed at any time using **free_elements()**. Each track item usually references a metadata tree or the content of an audiovisual material. Combination of track items builds up a complete editing.

bool free_elements (ordered_track_item_list *) const

Frees the list returned by **elements()**. Returns *false* if an error occurred.

I_timecode *get_end_timecode ()

Returns the end timecode for this track. Returned pointer should be freed by the API user.

I_timecode *get_start_timecode ()

Returns the start timecode for this track. Returned pointer should be freed by the API user.

uint32_t get_track_id () const

Returns an integer uniquely identifying the current track within its material. However, two tracks originating from a different material may have the same **track_id()** value.

const wchar_t *get_track_name (size_t &) const

Returns the name of the current track in a Unicode UTF16 string. After calling this function, the *size_t* parameter will contain the length (in bytes) of the returned string. Returned pointer must not be deleted.

ordered_track_item_list *item_seek_eu (int64_t)

This function returns the list of **I_track_item** from this track located at the given edit unit. The list can be empty if the edit unit is beyond track's scope. On an *event_metadata* track the list may contain several track items; all other tracks will contain at most one track item. Each track item provides methods to read its content. Hence when the user wishes to read the data at a given edit unit, he must first seek to the desired location thanks to **item_seek_eu()** or **item_seek_timecode()**. Then, the **I_track_item** just retrieved must be cast into **I_source_clip**, **I_dm_segment** or **I_dm_source_clip** that will expose methods to read the metadata or audiovisual data.

ordered_track_item_list *item_seek_timecode (const I_timecode *)

This function returns the list of **I_track_item** from this track located at the given timecode. Timecode information can be retrieved from the function **I_generic_material::get_timecode()**. The list can be empty if the edit unit is beyond track's scope. On an *event_metadata* track the list may contain several track items; all other tracks will contain at most one track item. Each track item provides methods to read its content. Hence when the user wishes to read the data at a given edit unit, he must first seek to the desired location thanks to **item_seek_eu()** or **item_seek_timecode()**. Then, the **I_track_item** just retrieved must be cast into **I_source_clip**, **I_dm_segment** or **I_dm_source_clip** that will expose methods to read the metadata or audiovisual data. Furthermore, data from different tracks read at the same timecode is synchronized and is meant to be played together.

int64_t origin ()

Returns the origin of this track measured in edit units. Future references to the track's content will be performed thanks to an offset relative to the origin.

bool remove_metadata (I_metadata_material *)

Removes the metadata material and its associated metadata tree if the current track is an event or static metadata track. Returns *false* if an error occurred.

bool set_track_name (const wchar_t *, size_t)

Sets the name of this track. The first parameter contains the **Unicode** UTF16 string and the second one its length in bytes. The *wchar_t** pointer can be destroyed after calling this function.

int64_t timecode_to_edit_unit (const I_timecode *)

Converts the timecode value to an edit unit on this track. Returns -1 if the edit unit cannot be computed.

track_type type () const

Returns the type of this track (*picture*, *sound*, *data*, *timeline_metadata*, *event_metadata* or *static_metadata*).

const char *type_name () const

Returns the type of this track in a string. Returned pointer must not be deleted.

Related Documentation

enum track_type

picture
sound
data
timeline_metadata
event_metadata
static_metadata
track_type_error

This enumeration is used to define the variety of **I_track** :

- **picture**: video track. These tracks may only contain **I_source_clip**.
- **sound**: sound track. These tracks may only contain **I_source_clip**.
- **data**: auxiliary data track (subtitles, etc.). These tracks may only contain **I_source_clip**.
- **timeline_metadata**: descriptive metadata track where track items are butted to each other. This kind of track may contain several **I_dm_source_clip** or **I_dm_segment**. Possibilities of track items concatenations are limited by strong constraints: **I_dm_source_clip** and **I_dm_segment** must be adjacent; one and only one must always be active at any instant on the track. These constraints are handled internally by MXFTk so that the items only need to be appended one by one and they will be time-linearly arranged automatically.
- **event_metadata**: descriptive metadata track where track items define events occurring at a given time and for a given duration along this track. **I_dm_source_clip** and **I_dm_segment** can be positioned without

constraints on this kind of track (i.e. they can overlap or some parts of the track can remain empty).

- **static_metadata**: descriptive metadata track where track items do not carry time positioning information. Metadata from this kind of track applies to the entirety of the material they reference. Only **I_dm_source_clip** and **I_dm_segment** are allowed on this kind of track.
- **track_type_error**: invalid track type detected.

4.2.1 I_track_item Class Reference

```
#include <I_track_item.hpp>
```

According to the MXF terminology, a track can be defined as an *ordered* list of objects containing audiovisual data, editing information or descriptive metadata. The indivisible segment of data from an **I_track** interface is an **I_track_item**. This pure virtual class has three possible derivations, each of them specifying the properties of the track at a given time. Depending on the track's type, different constructions of **I_track_item** are allowed:

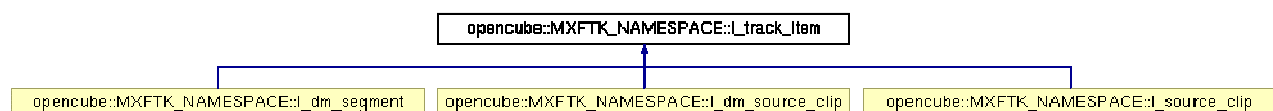
- On a timeline track (*picture*, *sound*, *data*), only editing information is allowed. Still, a difference remains if the track is found in an **I_generic_material** or in an **I_concrete_material**. In a concrete material the binary data stream is stored, while in a generic material the concrete materials' tracks are referenced and assembled all together to build an edition. Editions are represented by a combination of track items. As a result, a timeline track shall only contain **I_source_clip** objects. They are adjacent to each other in order to build a fully time-linear track representing a video, sound or data play out. **I_source_clip** may not overlap and for the entire duration of the track, one and only one **I_source_clip** should be active at a given edit unit.
- On a metadata timeline track (*timeline_metadata*) only metadata or metadata editing are allowed. Consequently, this kind of **I_track** shall only contain **I_dm_source_clip** or **I_dm_segment**. Once again, the same positioning constraints remain: **I_dm_source_clip** or **I_dm_segment** may not overlap and for the entire duration of the track, one and only one of them should be active at a given edit unit.
- On an *event_metadata* track, only metadata describing events occurring as the file is played is allowed. Consequently, this kind of track shall only contain **I_dm_segment**. Furthermore, positioning constraints disappear, these "segments" may overlap and "slices" of the track may remain empty.
- On a *static_metadata* track, only metadata applying to the entire duration of the linked essence tracks is allowed. Consequently, this kind of track shall only contain **I_dm_segment**. Additionally, these "segments" do not carry time constraints.

MXFTk user does not require a strong knowledge of these properties. The API ensures the consistency of each material and track. However, the user is responsible for the choice of the appropriate track to be created when adding metadata. Therefore, it is recommended to keep in mind the differences between *timeline_metadata*, *event_metadata* and *static_metadata* tracks.

The **I_track_item** class and its three possible derivations are described hereafter. The type of an **I_track_item** instance can be determined by performing a dynamic cast thanks to the following functions (for stability reasons, no direct calls to **dynamic_cast** shall be performed on MXFTk interfaces). Failure to do so would systematically lead to MXFTk's data structures corruption.

```
I_source_clip *source_clip_cast (I_track_item *)
I_dm_source_clip *dm_source_clip_cast (I_track_item *)
I_dm_segment *dm_segment_cast (I_track_item *)
```

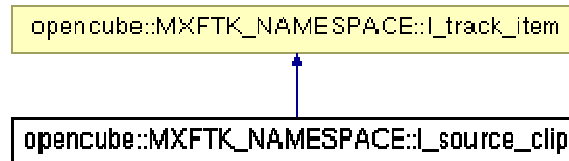
These functions will return NULL when trying to cast in the wrong type.



4.2.2 I_source_clip Class Reference

```
#include <I_track_item.hpp>
```

This class holds editing information for audio, video or data tracks. No metadata can be found in an **I_source_clip**. The play out of a track is described by a succession of **I_source_clip**. An **I_source_clip** references data contained in an **I_concrete_track**.



Public Member Functions

```

uint64_t current_eu_size ()
int64_t get_current_eu ()
I_timecode *get_current_timecode ()
int64_t get_duration () const
int64_t get_end_position () const
I_timecode *get_end_timecode () const
size_t read_eu (uint8_t *, size_t)
I_generic_material *get_referenced_material () const
const I_umid *get_referenced_material_umid () const
I_track *get_referenced_track () const
int64_t get_start_position () const
I_timecode *get_start_timecode () const
bool seek_eu (int64_t)
bool seek_next_eu ()
bool seek_previous_eu ()
  
```

Member Functions Documentation

uint64_t current_eu_size ()

Returns the size of the current edit unit from the referenced track. This function can be called to set the size of the buffer used when calling **read_eu()**. Return **UINT64_ERROR** if the size is unknown (usually when the current edit unit is not valid).

int64_t get_current_eu ()

Returns the current edit unit offset relative to the origin of the referenced track (not relative to the origin of the track containing this source clip). Return **INT64_ERROR** if the current edit unit is invalid. Returned value is comprised between **get_start_position()** and **get_end_position()**.

I_timecode *get_current_timecode ()

Returns the timecode corresponding to the current edit unit. This is a timecode relative to the track containing this source clip (not relative to the referenced track). Returned timecode is always comprised between **get_start_timecode()** and **get_end_timecode()**. Returned pointer must be deleted by MXFTk user. Return NULL if the current timecode could not be retrieved.

int64_t get_duration () const

Retrieves the duration of this **I_source_clip** (measured in edit units of the track containing this source clip). The sum of the duration of each source clip defines the total duration of the track. An edit unit is the time unit for the current track (usually a frame or a field for a video track). Please refer to [377M] for further information. The edit rate is the number of edit units per second. Note that the track containing the source clip may have an edit rate

different from the referenced track. Returns -1 if the duration is unknown.

int64_t get_end_position () const

Returns the offset of the last edit unit to be read on the referenced track. The offset is relative to the origin of the referenced track. Return *INT64_ERROR* if this information could not be retrieved (notably if the duration of the source clip is unknown).

I_timecode *get_end_timecode () const

Returns the timecode corresponding to the last edit unit of this source clip. This is a timecode relative to the track containing this source clip (not relative to the referenced track). Return NULL if this timecode could not be retrieved.

size_t read_eu (uint8_t *, size_t)

Reads the data from the current edit unit and stores it in the *uint8_t** buffer. The *size_t* parameter specifies the size of the buffer. Usually, it is set to the size returned by *current_eu_size()*. However, the reading of an edit unit can be split in several calls to *read_eu()* with smaller buffer sizes if required. For instance if *current_eu_size()* returns a size of 200000 bytes. Calling *read_eu(buffer, 200000)* or calling twice *read_eu(buffer, 100000)* will retrieve all the data from the current edit unit.

I_generic_material *get_referenced_material () const

Returns a pointer (reference) to the material containing the *I_track* referenced by *get_referenced_track()*. Return NULL if no material is being referenced or if the material is not in the file. Returned material can be cast into an *I_concrete_material* if this source clip is in an *I_generic_material*.

const I_umid *get_referenced_material_umid () const

Returns the Unique Material Identifier of the referenced material. This function is particularly useful when *get_referenced_track()* returns NULL. If the UMID is not nil, it generally means that the referenced material is stored in another MXF file (this can notably be the case when reading OpAtom files). Retrieving the UMID of the missing material may help to locate it in a database environment.

I_track *get_referenced_track () const

Returns a pointer (reference) to the track containing the audiovisual source to be edited on the current track. Return NULL if no track is being referenced or if the referenced material is not in the file. Returned track can be cast into an *I_concrete_track* if this source clip is in an *I_generic_material*.

int64_t get_start_position () const

Returns the offset of the first edit unit to be read on the referenced track. The offset is relative to the origin of the referenced track. Returns *INT64_ERROR* if this information could not be retrieved.

I_timecode *get_start_timecode () const

Returns the timecode corresponding to the first edit unit of this source clip. This is a timecode relative to the track containing this source clip (not relative to the referenced track). Returns NULL if this timecode could not be retrieved.

bool seek_eu (int64_t)

Seeks a given edit unit to be read on the referenced track. The edit unit must be comprised between *get_start_position()* and *get_end_position()*. Returns *false* if the edit unit can not be reached.

bool seek_next_eu ()

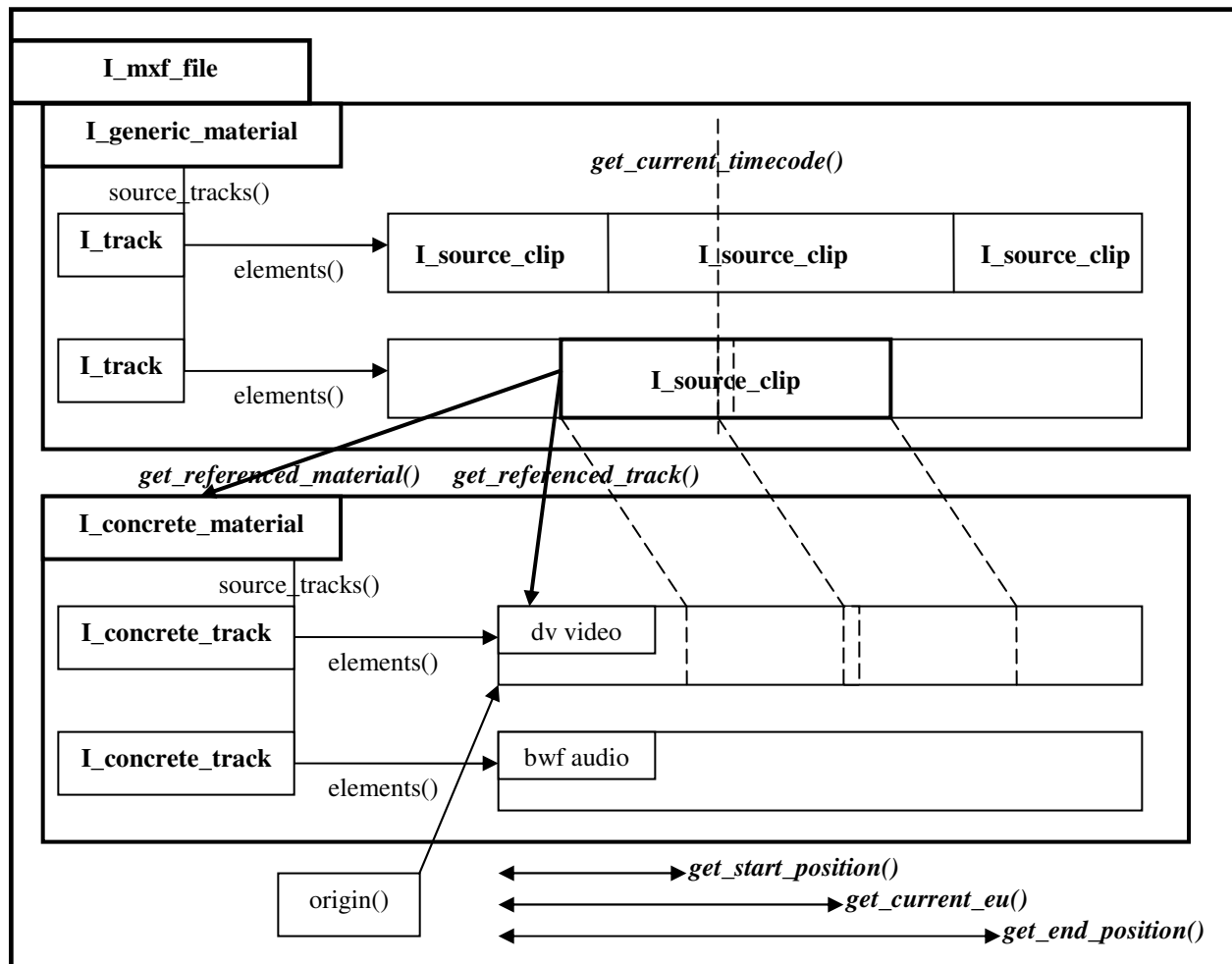
Jumps to the next edit unit to be read on the referenced track. Returns *false* if the edit unit can not be reached.

bool seek_previous_eu ()

Jumps to the previous edit unit to be read on the referenced track. Returns *false* if the edit unit can not be reached.

I_source_clip data access

The following figure illustrates data access from the source clip.

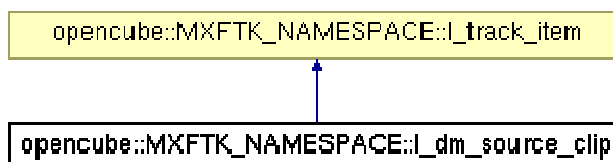


The data carried by an **I_source_clip** is the content of the source data track returned by `get_referenced_track()`, starting from the offset returned by `get_start_position()` and ending at the offset returned by `get_end_position()`.

4.2.3 I_dm_source_clip Class Reference

```
#include <I_track_item.hpp>
```

This class holds editing information for descriptive metadata to be found on *timeline_metadata* tracks. It references metadata contained in other materials. Within an MXF file, metadata tracks from an **I_generic_material** (MXF terminology: material package) can reference tracks from an **I_concrete_material** (MXF terminology: top-level source package) or even from a low-level source package. The current version of MXFTk does not allow addition of **I_dm_source_clip** to a metadata track. However, they will be recognized if found while decoding a file.



Public Member Functions

```
track_list *get_documented_tracks () const
int64_t get_duration () const
int64_t get_end_position () const
I_timecode* get_end_timecode () const
int64_t get_start_position () const
I_timecode* get_start_timecode () const
I_generic_material *get_referenced_material () const
const I_umid *get_referenced_material_umid () const
I_track *get_referenced_track () const
```

Member Functions Documentation

track_list *get_documented_tracks () const

Returns a list of *picture*, *sound* or *data* tracks from the material containing the current metadata track. Metadata from this **I_dm_source_clip** documents the tracks from this list.

int64_t get_duration () const

Retrieves the duration of this **I_dm_source_clip** (measured in edit units of the track containing this dm source clip). The sum of the duration of each metadata source clip and metadata segment defines the total duration of the track. An edit unit is the time unit for the current track. Please refer to [377M] for further information. The edit rate is the number of edit units per second. Note that the track containing this metadata source clip may have an edit rate different from the referenced track. Returns -1 if the duration is unknown.

int64_t get_end_position () const

Returns the offset of the last edit unit to be read on the referenced track. The offset is relative to the origin of the referenced track. Returns *INT64_ERROR* if this information could not be retrieved.

I_timecode* get_end_timecode () const

Returns the timecode corresponding to the last edit unit of this metadata source clip. This is a timecode relative to the track containing this metadata source clip (not relative to the referenced track). Returns NULL if this timecode could not be retrieved.

I_generic_material *get_referenced_material () const

Returns a pointer (reference) to the material containing the **I_track** referenced by **get_referenced_track()**. Return NULL if no material is being referenced or if the material is not in the file. Returned material can be cast into an **I_generic_material** if this source clip is in an **I_generic_material**.

const I_umid *get_referenced_material_umid () const

Returns the Unique Material Identifier of the referenced material. This function is particularly useful when **get_referenced_track()** returns NULL. If the UMID is not nil, it generally means that the referenced material is stored in another MXF file (this can notably be the case when reading OpAtom files). Retrieving the UMID of the missing material may help to locate it in a database environment.

I_track *get_referenced_track () const

Returns a pointer (reference) to the track containing the audiovisual source to be edited on the current track. Returns NULL if no track is being referenced or if the referenced material is not in the file. Returned track can be cast into an **I_concrete_track** if this source clip is in an **I_generic_material**.

int64_t get_start_position () const

Returns the offset of the first edit unit to be read on the referenced track. The offset is relative to the origin of the referenced track. Returns *INT64_ERROR* if this information could not be retrieved.

I_timecode *get_start_timecode () const

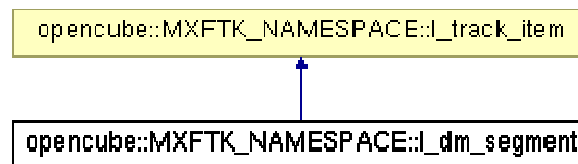
Returns the timecode corresponding to the first edit unit of this metadata source clip. This is a timecode relative to the track containing this metadata source clip (not relative to the referenced track). Returns NULL if this timecode could not be retrieved.

The metadata carried by an **I_dm_source_clip** is the content of the metadata track returned by **get_referenced_track()**, starting from the offset returned by **get_start_position()** and ending at the offset returned by **get_end_position()**. The metadata from this **I_dm_source_clip** documents the tracks returned by **get_documented_tracks()**.

4.2.4 I_dm_segment Class Reference

```
#include <I_track_item.hpp>
```

This class holds descriptive metadata. Each **I_dm_segment** references an **I_metadata_material** as well as a list of tracks documented by this material. Please, refer to the **I_metadata_material** class reference for further information on how to create and extract metadata.



Public Member Functions

track_list *get_documented_tracks () const

int64_t get_duration () const

I_timecode *get_end_timecode () const

I_metadata_material *get_metadata_material () const

I_timecode *get_start_timecode () const

Member Functions Documentation

track_list *get_documented_tracks () const

Returns a list of *picture*, *sound* or *data* tracks from the material containing the current metadata track. Metadata from this **I_dm_source_clip** documents the tracks from this list.

int64_t get_duration () const

Retrieves the duration of this **I_dm_segment** (measured in edit units of the track containing this metadata segment). The sum of the duration of each metadata source clip and metadata segment defines the total duration of the track. An edit unit is the time unit for the current track. Please refer to [377M] for further information. The edit rate is the number of edit units per second. Note that the track containing this metadata source clip may have an edit rate different from the referenced track. Returns -1 if the duration is unknown.

I_timecode *get_end_timecode () const

Returns the timecode corresponding to the last edit unit of this metadata segment. This is a timecode relative to the track containing this metadata segment. Returns NULL if this timecode could not be retrieved.

I_metadata_material *get_metadata_material () const

Returns the **I_metadata_material** containing the descriptive metadata tree and the time positioning information for this **I_dm_segment**.

`I_timecode *get_start_timecode () const`

Returns the timecode corresponding to the first edit unit of this metadata segment. This is a timecode relative to the track containing this metadata segment. Returns NULL if this timecode could not be retrieved.

5. Concrete Material

Concrete material and tracks enable the manipulation of audiovisual data. They grant access to the “binary” video or audio streams of each track thanks to read functions. Concrete materials have a UMID (SMPTE Unique Material Identifier, see [330M]). MXFTk user can set its value after creating the material or leave the default one automatically set by the API.

5.1 I_concrete_material Class Reference

```
#include <I_concrete_material.hpp>
```

This class contains the audiovisual data embedded in an MXF file. **I_generic_material**’s tracks describe an editing of one or several **I_concrete_material**’s tracks in order to build the file play out.

```
opencube::MXFTK_NAMESPACE::I_generic_material
```



```
opencube::MXFTK_NAMESPACE::I_concrete_material
```

Public Member Functions

```
void compute_duration ()
bool end_of_stream (unsigned int)
wrapping get_wrapping () const
size_t write (unsigned int, const uint8_t *, size_t)
```

Constructor and Destructor Documentation

bool NewConcreteMaterialInterface (I_concrete_material **, locators *, wrapping, rational*, bool)

Retrieves a pointer on an **I_concrete_material** interface. The second parameter is a list of complete paths to the audio or video source files to be embedded in this concrete material. This can be the path to a dv, mpg, wav, etc. file or to a series of still images (j2k, bmp, tif) to be wrapped as a single track. In that case the syntax to define the series is the following one: in order to designate a series of images the following syntax should be inserted in the path of the locator `#[d, f, l, s]` where:

- d should be a number designating the number of digits
- f should be the first number of the file to be wrapped
- l should be the last number of the file to be wrapped
- s should be the step between two successive files to be wrapped

Note that it is perfectly valid to set a negative step. For instance

"C:\images\myfile#[4, 10, 500, 2].jp2", "C:\sound\sound.wav" will perform the wrapping of the following series of files:

C:\images\myfile0010.jp2

C:\images\myfile0012.jp2

C:\images\myfile0014.jp2

...

C:\images\myfile0498.jp2

C:\images\myfile0500.jp2

with the audio wave file "sound.wav"

The third parameter specifies the wrapping of this material. The fourth parameter can be used to set the default edit rate (the edit rate that will be used when its value cannot be computed from the essence – typically when wrapping audio). Finally the fifth parameter can be set to *true* to force MXFTk to ignore the audio from the DV files that will be wrapped. In that case, no audio track will be created for the DV streams. Returns *false* if the allocation failed.

bool NewExtConcreteMaterialInterface (I_concrete_material **, const char *, rational *, bool)

Retrieves a pointer on an **I_concrete_material** interface that will contain an external reference to a raw media file. The second parameter is the path to the audio or video source file to be referenced by the material. The third parameter can be used to set the default edit rate (the edit rate that will be used when its value cannot be computed from the essence – typically when wrapping audio). Finally the fourth parameter can be set to *true* to force MXFTk to ignore the audio from the DV streams that will be wrapped. In that case, no audio track will be created for the DV streams. Returns *false* if the allocation failed.

bool NewCryptedMaterialInterface (I_concrete_material **, crypted_locators *, rational *, bool)

Retrieves a pointer on an **I_concrete_material** interface. The second parameter is a list of complete paths (and crypt keys) to the audio or video source files to be embedded and crypted in this concrete material. The third parameter can be used to set the default edit rate (the edit rate that will be used when its value cannot be computed from the essence – typically when wrapping audio). Finally the fourth parameter can be set to *true* to force MXFTk to ignore the audio from the DV streams that will be wrapped. In that case, no audio track will be created for the DV streams. Crypted content is always frame wrapped. Returns *false* if the allocation failed.

bool NewStreamMaterialInterface (I_concrete_material **, I_essence_stream_task *, wrapping, bool)

Retrieves a pointer on an **I_concrete_material** interface initialized with audio or video streams. This function should be used when wrapping an MXF file on the fly as the audiovisual material is received from a streaming media. The **I_essence_stream_task** (or **I_crypted_essence_stream_task**) provides control over the audiovisual stream. User should refer to the “Streaming” chapter for a complete overview of MXFTk’s streaming capabilities. The third parameter specifies the wrapping of this material. Finally the fourth parameter can be set to *true* to force MXFTk to ignore the audio from the DV streams that will be wrapped. In that case, no audio track will be created for the DV streams. Returns *false* if the allocation failed.

bool FreeConcreteMaterialInterface (I_concrete_material **)

Frees an **I_concrete_material** interface. Returns *false* if the deallocation failed.

Member Functions Documentation

void compute_duration ()

Use this function if you need to know the duration of your material before attaching it to an MXF file. It will force the computation of the duration of the tracks from this material. However, be aware that this task may be time-consuming.

bool end_of_stream (unsigned int)

States that an audiovisual data stream is now closed. This function will be operative only if this material was created using **NewStreamMaterialInterface()**. After **end_of_stream()**, future calls to the **write()** function on this stream will not operate. Calls to this function should be performed only within the function **I_essence_stream_task::data_request** to notify to MXFTk that no more data is to be received from the stream. The integer parameter is a unique identifier of the current stream, also parameter of the function **I_essence_stream_task::data_request**.

wrapping get_wrapping () const

Gets the wrapping of the current material.

void write (unsigned int, const uint8_t *, size_t)

Supplies the current material with a buffer originating from an audiovisual data stream. This function will be operative only if this material was created using **NewStreamMaterialInterface()**. Calls to this function should be performed only within the function **I_essence_stream_task::data_request**. The integer parameter is a unique identifier of the current stream, also parameter of the function **I_essence_stream_task::data_request**. The second is the buffer of data and the third one the size in bytes of this buffer.

Related Documentation

enum wrapping

std_wrapping
clip_wrapping
frame_wrapping
line_wrapping
custom_wrapping
evtr_wrapping
xdcam_wrapping
p2_wrapping
k2_wrapping
opzero_wrapping
dcp_wrapping
wrapping_error

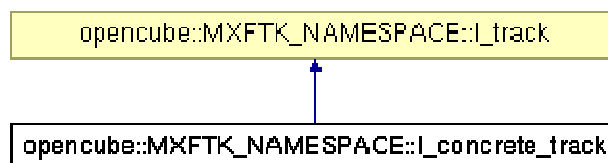
This enumeration defines the wrapping of an essence embedded in an MXF file (*std_wrapping* defines the default wrapping for this essence type). Some wrappings are not allowed in particular cases. MXFTk currently does not support *line_wrapping*. You must create your materials in *xdcam_wrapping* in order to attach them to an **I_xdcam_dv_file**, **I_xdcam_imx_file** or **I_xdcam_proxy_file**; *evtr_wrapping* in order to attach them to an **I_evtr_file** file; *p2_wrapping* in order to attach them to a P2 file; *k2_wrapping* in order to build files similar to the ones produced by a Thomson Grass Valley K2 server; *opzero_wrapping* in order to attach them to an **I_op1a_file** built with the OpZero constructor; *dcp_wrapping* in order to attach them to an **I_dcp1_file**.

5.2 I_concrete_track Class Reference

```
#include <I_concrete_track.hpp>
```

This class represents the track of an **I_concrete_material**. It contains the “physical” audiovisual data and supplies methods to access its content. This class directly accesses the audiovisual data of the file. When reading the output of an MXF file, this class should not be used; instead prefer the classes **I_source_clip** of the tracks from the generic material which holds all the editing information

Concrete tracks can be retrieved thanks to a call to **source_tracks()** from the inherited class **I_generic_material**. Items from the returned list can be directly cast into **I_concrete_track** (do not use *dynamic_cast*, this would cause crashes).



Public Member Functions

```

uint64_t current_eu_size ()
bool free_descriptors (metadata_list *)
const uint8_t *get_cryptographic_key_id ()
int64_t get_current_eu ()
uint64_t get_current_offset()
metadata_list *get_descriptors () const
I_essence_type *get_essence_type ()
const char *get_locator () const
size_t read_eu (uint8_t *, size_t s)
bool seek_eu (int64_t)
bool seek_next_eu ()
  
```



```
bool seek_previous_eu ()
bool seek_timecode (const I_timecode *)
void set_cipher_key (const uint8_t *)
void set_external_ref_file (const char *)
```

Member Functions Documentation

uint64_t current_eu_size ()

Returns the size in bytes of the current edit unit. This function can be used to set the size of the buffer passed to the function `read_eu()`. Returns 0 if all the data for this edit unit has been read or if an error occurred.

bool free_descriptors (metadata_list *)

Frees the list returned by `get_descriptors()`.

const uint8_t *get_cryptographic_key_id ()

Call this function to know if the concrete track is protected by an key. This will notably be the case when manipulating files following the standard of the Digital Cinema Initiative. The function returns the 16-byte long public key protecting the track, otherwise it will return NULL if the track's content is not encrypted. You need to provide the corresponding cipher key in order to be able to read the content of this track.

int64_t get_current_eu ()

Returns the current edit unit relative to the origin of the track. The real position within the embedded audiovisual data is actually the sum of the track's origin with this edit unit. For instance on a video track, if the origin is 5 and the current edit unit 10, when calling `read_eu()`, the 15th frame will be read. The legal edit units range is `[-track_origin, -track_origin+track_duration-1]`. Returns `INT64_ERROR` if an error occurred.

uint64_t get_current_offset()

Returns the current offset in bytes from the beginning of the file. Returns `UINT64_ERROR` if an error occurred.

metadata_list *get_descriptors () const

Retrieves a list of `I_metadata` describing the nature and the properties of the track's content. Usually the list will contain a single item. These properties are specific to each essence type and can be checked with the MXF File Format glossary [377M]. The descriptors grant access to properties such as the size or the aspect ratio of images from a video track. Some of the information found in the descriptors will be redundant with the one provided by `get_essence_type()`.

I_essence_type *get_essence_type ()

Retrieves an `I_essence_type` interface describing the type of essence embedded in this concrete track. MXFTk user can use this function to figure out if the track contains DV, MPEG, D10, etc. data. When possible it also states if it is a NTSC or PAL source.

const char *get_locator () const

Returns a string identifying the original source (the path to the file used to build this track) of the track's content. This function is only useful when building concrete material from several source files. The call to this function lets you know which of these files the current track embeds. Returned pointer must not be deleted.

size_t read_eu (uint8_t *, size_t s)

Fills a buffer with the binary audiovisual data contained in this track at the current edit unit. This function will fill the buffer only when reading an MXF file (not while creating one)! The second parameter states the size of the buffer provided and the returned value indicates the number of bytes that were effectively read from the track. The current edit unit can be entirely read through successive calls to this function until it returns a nil value.

bool seek_eu (int64_t)

Puts the reading pointer of the track on the given edit unit. Returns *false* if this task could not be performed.

bool seek_next_eu ()

Puts the reading pointer of the track on the next edit unit. Returns *false* if this task could not be performed.

bool seek_previous_eu ()

Puts the reading pointer of the track on the previous edit unit. Returns *false* if this task could not be performed.

bool seek_timecode (const I_timecode *)

Puts the reading pointer of the track on the timecode value specified. Timecode information can be retrieved from the **I_timecode_material** of the **I_concrete_material** containing this **I_concrete_track**. Returns *false* if this task could not be performed.

void set_cipher_key (const uint8_t *)

Call this function to set the private 16-byte long cipher key corresponding to the public cryptographic key protecting this track. If the key is correct, you will now be able to read the content of this track.

void set_external_ref_file (const char *)

Sets the path to the external reference for this concrete track. Call this function to load the referenced file when reading an MXF file with external references.

5.3 I_essence_type Class Reference

```
#include <I_essence_type.hpp>
```

This class provides information on the content of a concrete track such as the video or sound format. It should be used to find out which decoder is required for playing the audiovisual data (essence) embedded on a track. Please refer to SMPTE 377M for further information on the values returned by this class.

Public Member Functions

```
rational *aspect_ratio () const
uint32_t bit_rate () const
uint32_t component_depth () const
uint32_t display_height () const
uint32_t display_width () const
electro_spatial_form *electro_spatial_formulation () const
locators *external_references (bool& ) const
layout frame_layout() const
essence_format get_format () const
essence_source get_type () const
uint32_t nb_channels () const
uint32_t quantization () const
sample_structure sampling () const
rational *sampling_rate () const
uint32_t stored_height () const
uint32_t stored_width () const
bool update_external_references (locators *, bool)
```

Member Functions Documentation

rational *aspect_ratio () const

Returns the aspect ratio of this essence if this information can be retrieved (e.g. 16/9 or 4/3). Return NULL if the aspect ratio is not defined for this kind of essence or if this information could not be retrieved. Returned pointer should not be deleted.

uint32_t bit_rate () const

Returns the bit rate per second in Mbps. Returns 0 if the bit rate is not defined for this kind of essence or if this information could not be retrieved.

uint32_t component_depth () const

Returns the component depth (e.g. 8, 10 or 16) if applicable (video track).

uint32_t display_height () const

Returns the display height in pixels (video track). Returns 0 if the height is not defined for this kind of essence or if this information could not be retrieved.

uint32_t display_width () const

Returns the display width in pixels (video track). Returns 0 if the width is not defined for this kind of essence or if this information could not be retrieved.

electro_spatial_form *electro_spatial_formulation () const

Returns the electro spatial formulation if applicable (audio track). Please refer to SMPTE 377M for further information.

locators *external_references (bool&) const

Returns the list of external references for the concrete track referenced by this essence type object. The boolean value is set to *true* if the returned locators are URLs. If set to *false* the returned locators are just textual information.

layout_frame_layout() const

Returns the frame layout (interlaced, progressive, etc.) if applicable (video track). Please refer to SMPTE 377M for further information.

essence_format get_format () const

Returns the essence format when this data can be retrieved. PAL, NTSC, interlaced, etc.

essence_source get_type () const

Returns the essence type. MPEG, DV, AES, BWF, etc.

uint32_t nb_channels () const

Returns the number of channels (audio track).

uint32_t quantization () const

Returns the quantization in bits per sample (audio track).

sample_structure sampling () const

Returns the sample rate for this essence (if applicable).

rational *sampling_rate () const

Returns the sampling rate of this essence if this information can be retrieved (audio track). Return NULL if the aspect ratio is not defined for this kind of essence or if this information could not be retrieved. Returned pointer should not be deleted.

uint32_t stored_height () const

Returns the stored height in pixels (video track). Returns 0 if the height is not defined for this kind of essence or if this information could not be retrieved.

uint32_t stored_width () const

Returns the stored width in pixels (video track). Returns 0 if the width is not defined for this kind of essence or if this information could not be retrieved.

bool update_external_references (locators *, bool)

Sets the new location of the externally referenced files. The boolean value should be set to *true* if the locators are URLs or *false* if they are just textual information. This function is particularly useful for updating a file when doing a partial restore of an MXF file with external references or when updating an MXF file with external references that moved. Returns *false* if an error occurred.

Related Documentation

enum essence_source

<i>ess_unknown</i>	// unrecognized essence
<i>ess_d10</i>	// D10
<i>ess_d11</i>	// D11
<i>ess_dv_unknown</i>	// DV other than IEC or SMPTE
<i>ess_dv_iec</i>	// DV IEC
<i>ess_dv_cam_iec</i>	// DV IEC from dvcam-1
<i>ess_dv_smpTE</i>	// DV SMPTE
<i>ess_mpeg_es</i>	// MPEG Elementary Stream
<i>ess_mpeg_pes</i>	// MPEG Packetized Elementary Stream
<i>ess_mpeg_ps</i>	// MPEG Program Stream
<i>ess_mpeg_ts</i>	// MPEG Transport Stream
<i>ess_mpeg4</i>	// MPEG4 Stream
<i>ess_uncompressed_unknown</i>	// Uncompressed other than sd or hd
<i>ess_uncompressed_sd</i>	// Uncompressed SD
<i>ess_uncompressed_hd1080</i>	// Uncompressed HD 1080 lines
<i>ess_uncompressed_hd720</i>	// Uncompressed HD 720 lines
<i>ess_jpeg2k</i>	// JPEG 2000
<i>ess_bwf</i>	// Audio Broadcast Wave
<i>ess_aes3</i>	// Audio AES3
<i>ess_a_law</i>	// Audio A-law
<i>ess_aiff</i>	// Audio AIFF
<i>ess_vc3</i>	// VC3 Video

enum essence_format

<i>for_unknown</i>	// unrecognized format
<i>for_525_5994p</i>	// NTSC 59.94Hz progressive
<i>for_525_5994i</i>	// NTSC 59.94Hz interlaced
<i>for_525_60i</i>	// NTSC 60Hz interlaced
<i>for_625_50i</i>	// PAL 50Hz interlaced
<i>for_625_50p</i>	// PAL 50Hz progressive
<i>for_720_2398p</i>	// HD 720 lines 23.98Hz progressive
<i>for_720_24p</i>	// HD 720 lines 24Hz progressive
<i>for_720_25p</i>	// HD 720 lines 25Hz progressive
<i>for_720_2997p</i>	// HD 720 lines 29.97Hz progressive
<i>for_720_30p</i>	// HD 720 lines 30Hz progressive
<i>for_720_50p</i>	// HD 720 lines 50Hz progressive
<i>for_720_5994p</i>	// HD 720 lines 59.94Hz progressive
<i>for_720_60p</i>	// HD 720 lines 60Hz progressive
<i>for_1080_2398p</i>	// HD 1080 lines 23.98Hz progressive
<i>for_1080_2398sf</i>	// HD 1080 lines 23.98Hz segmented frame
<i>for_1080_24p</i>	// HD 1080 lines 24Hz progressive
<i>for_1080_24sf</i>	// HD 1080 lines 24Hz segmented frame
<i>for_1080_25p</i>	// HD 1080 lines 25Hz progressive
<i>for_1080_25sf</i>	// HD 1080 lines 25Hz segmented frame
<i>for_1080_2997p</i>	// HD 1080 lines 29.97Hz progressive


```
for_1080_2997sf // HD 1080 lines 29.97Hz segmented frame  
for_1080_30p // HD 1080 lines 30Hz progressive  
for_1080_30sf // HD 1080 lines 30Hz segmented frame  
for_1080_50i // HD 1080 lines 50Hz interlaced  
for_1080_50p // HD 1080 lines 50Hz progressive  
for_1080_5994p // HD 1080 lines 59.94Hz progressive  
for_1080_5994i // HD 1080 lines 59.94Hz interlaced  
for_1080_60i // HD 1080 lines 60Hz interlaced  
for_1080_60p // HD 1080 lines 60Hz progressive
```

enum sample_structure

```
samp_unknown  
samp_4_1_1  
samp_4_2_0  
samp_4_2_2  
samp_4_4_4_4
```

enum layout

```
layout_full_frame  
layout_separate_fields  
layout_single_field  
layout_mixed_fields  
layout_segmented_frame  
layout_unknown
```

enum electro_spatial_form

```
formulation_two_channel_default  
formulation_two_channel  
formulation_single_channel  
formulation_primary_secondary  
formulation_stereophonic  
formulation_single_channel_double_frequency  
formulation_stereo_left_channel_double_frequency  
formulation_stereo_right_channel_double_frequency  
formulation_multi_channel_default  
formulation_unknown
```

6. Metadata Material

Metadata material enables the manipulation of descriptive metadata either built by MXFTk user or read from an MXF file. In some cases, this structure is also used to read specific structural metadata (such as descriptors from a concrete material or identification sets). Metadata (**I_metadata**) carries a set of properties. A property **I_property** is defined by a **label_c**, normalized and validated thanks to a dictionary, and a value **I_value**. A label is a *char** string and a value is a memory pointer of a given type that can contain metadata (when the pointer is referencing an **I_metadata** interface). Using this mechanism of metadata objects referencing other metadata objects a tree structure can be easily built. Only MXFTk Advanced version will let you attach descriptive metadata to your MXF files.

6.1 I_metadata_material Class Reference

```
#include <I_metadata_material.hpp>
```

This class embeds a metadata tree and time positioning information. In order to tie an **I_metadata_material** to a metadata track, it is compulsory to use the function **add_metadata()** from the interface **I_track**.

```
opencube::MXFTK_NAMESPACE::I_generic_material
```



```
opencube::MXFTK_NAMESPACE::I_metadata_material
```

Public Member Functions

```
wchar_t * get_comment (size_t &) const
int64_t get_duration () const
I_metadata * get_metadata () const
metadata_type get_metadata_type () const
int64_t get_start_position () const
bool replace_metadata (I_metadata *)
```

Constructor and Destructor Documentation

```
bool NewMetadataMaterialInterface (I_metadata_material **, metadata_type, I_metadata *, int64_t,
int64_t, const wchar_t *, size_t)
```

Retrieves a pointer on an **I_metadata_material** interface. The second parameter sets the type of this metadata (*m_timeline*, *m_event* or *m_static*). The third parameter is an interface pointing to the metadata tree to be embedded. The fourth and fifth parameters respectively set the offset and the duration (measured in edit units of the targeted metadata track – leave to -1 to have MXFTk compute it automatically when possible) of this material in the metadata track. Finally the last two parameters respectively points to a Unicode UTF16 string and specifies its size in bytes. This string is meant to comment this new metadata material. It can be NULL and can be deleted after calling this function. Returns *false* if the allocation failed.

Depending on the metadata type, the offset and duration data will be ignored:

- *m_timeline*: offset into the targeted track (fourth parameter) will be ignored.
- *m_event*: both will be used.
- *m_static*: both the offset and the duration into the targeted track (fourth and fifth parameters) will be ignored.

```
bool FreeMetadataMaterialInterface (I_metadata_material **)
```

Frees an **I_metadata_material** interface. Returns *false* if the deallocation failed.

Member Functions Documentation

wchar_t * get_comment (size_t &) const

Retrieves a Unicode UTF16 string commenting the current metadata material. It also sets its length in the `size_t` parameter. This string must not be deleted or edited.

int64_t get_duration () const

Gets the duration of this material in edit units. A negative value means that the duration is unknown or not applicable (in an *m_static* metadata material).

I_metadata *get_metadata () const

Retrieves a reference to the metadata tree embedded in this material. Returned pointer must not be deleted but can be edited.

metadata_type get_metadata_type () const

Returns the type of this metadata material (*m_timeline*, *m_event* or *m_static*).

int64_t get_start_position () const

Gets the offset relative to the origin of the metadata track that contains (or will contain) this metadata material. A negative value means that this offset is unknown or not applicable (in an *m_static* metadata material).

bool replace_metadata (I_metadata*)

Replaces the current descriptive metadata tree with a new one. The previous one will be automatically destroyed. Returns *false* if an error occurred.

Related Documentation

enum metadata_type

m_timeline
m_event
m_static

This enumeration defines the three possible types of metadata:

m_timeline: metadata to be written in or read from a *timeline_metadata* track. In this kind of track, metadata materials are added sequentially, one after the other. Metadata materials are adjacent, can not overlap and one and only one metadata material must be active at any time on the track.

m_event: metadata to be written in or read from an *event_metadata* track. In this kind of track, metadata materials are positioned at a given offset for a given duration. There are no constraints: metadata materials may overlap, have nil duration and some parts of the track may remain empty.

m_static: metadata to be written in or read from a *static_metadata* track. In this kind of track, no position or duration references are allowed. This metadata material applies to the entirety of the track.

6.2 I_metadata Class Reference

```
#include <I_metadata.hpp>
```

This class handles descriptive metadata. A metadata framework to be embedded in a metadata material is structured as a tree. An **I_metadata** contains a set of properties. Each property links a value of a given type to a **label_c** (a string normalized according to a dictionary). The value of a property can be an **I_metadata** as well. This gives the opportunity to build a tree of **I_metadata** interfaces: a root **I_metadata** interface may contain among its properties other instances of **I_metadata** and so on. This kind of construction is often required in the usual metadata schemes such as the DMS1. DMS1 let's you set for instance the location of the clip, the actors appearing in the scene, etc...

MXFTk's metadata scheme manages all the DMS1 metadata. Metadata that does not belong to this particular scheme is not handled with the default dictionary. In order to address other metadata schemes it is required to update the

dictionary thanks to the function **SetDictionary()**. The default dictionary is located in the directory `dic` of your installation. You will find there a folder named `DMS1` containing the XML files drawing an exhaustive list of descriptive metadata. Although the dictionary is used to validate the exactness of the metadata trees, it is recommended to have an elementary knowledge of the schemes addressed in [EG42] [380M] in order to build your own trees.

```
typedef const char* label_c
    String defining a metadata by its name.
```

Public Member Functions

```
bool add (const I_property *)
bool free_properties (properties_list *) const
properties_list * get_dic_properties () const
label_c get_label () const
properties_list * get_properties () const
bool output_xml (const char *)
bool remove (const I_property *)
```

Constructors and Destructor Documentation

```
bool NewMetadataInterfaceByLabel (I_metadata **, label_c)
```

Retrieves a pointer on an **I_metadata** interface. The second parameter specifies a label identifying the metadata. This label must strictly follow the dictionary naming convention (domain:property_name) such as “DMS1:SceneFramework” or even “file_format:Preface” when dealing with structural metadata. The **label_c** object can be destroyed after calling this function. Returns *false* if the allocation failed.

```
bool NewMetadataInterfaceByXML (I_metadata **, const char *)
```

Retrieves a pointer on an **I_metadata** interface. The second parameter specifies the complete path to an XML file describing a metadata tree. Returns *false* if the allocation failed.

Warning: the XML scheme must effectively match an arborescence structure: no more than one metadata can be a root. Failure to do so will invalidate the loading of the tree or lead to hectic behaviour.

```
bool FreeMetadataInterface (I_metadata **)
```

Frees an **I_metadata** interface. Returns *false* if the deallocation failed.

Member Functions Documentation

```
bool add (const I_property *)
```

Adds a property to the current metadata. The content of the property will be copied; therefore the **I_property** pointer can be freed after calling this function. Returns *false* if an error occurred.

Warning: if the current metadata already contains a property with the same **label_c** it will be destroyed and replaced by the new one. However, in the case of a property which type is **MXF_ARRAY_STRONG_REF** or **MXF_ARRAY_WEAK_REF**, the **I_metadata** interfaces of the property will be added to those already stored in the current metadata. This mechanism helps building a tree step-by-step without destroying previously created sub-trees.

```
bool free_properties (properties_list *) const
```

Frees the properties list returned by **get_properties()** or **get_dic_properties()**. Returns *false* if an error occurred.

```
properties_list * get_dic_properties () const
```

Returns a list referencing all the properties that can be added to the current metadata according to the dictionary. This function can be used to rapidly retrieve all the possible properties for the current node of the tree. Returned list must be freed using the function **free_properties()**.

label_c get_label () const

Gets a label identifying the current metadata.

properties_list * get_properties () const

Returns a list referencing all the properties from the current metadata. This is a read-only structure, meaning that modifying the list or its content will not modify the properties stored within the current metadata. Returned list must be deleted using **free_properties()**.

bool output_xml (const char *)

Dumps an XML representation of the current metadata tree. The XML file will be written at the location specified by the string. Returns *false* if an error occurred.

bool remove (const I_property *)

Destroys a property stored in the current metadata object. Return *false* if an error occurred.

Warning: The **label_c** of the property is compared to the **label_c** of the properties stored within the current metadata. If a **label_c** is matching then the internal property will be entirely destroyed. However if the property's type is **MXF_ARRAY_STRONG_REF** or **MXF_ARRAY_WEAK_REF** then only referenced **I_metadata** interfaces will be deleted. This mechanism helps destroying only some branches of the metadata tree.

Warning: The parameter (**I_property**) is not destroyed; it must be freed by MXFTk user.

6.3 I_property Class Reference

```
#include <I_property.hpp>
```

An **I_metadata** interface contains several **I_property**. A property is a label associated to a value of a given type.

Public Member Functions

label_c get_label () const

metadata_list *get_dic_metadata_sets () const

const I_value *get_value () const

bool free_metadata_sets (metadata_list *) const

Constructor and Destructor Documentation

bool NewPropertyInterface (I_property **, label_c, const I_value *)

Retrieves a pointer on an **I_property** interface. The second parameter is a label identifying the metadata according to the naming convention of the dictionary (DMS1:ClipFramework for instance). The third parameter is a reference to the property's value. This one should not be destroyed as long as the new property has not been deleted.

bool FreePropertyInterface (I_property **)

Frees an **I_property** interface. The **I_value** interface that was used to create this property (**I_value** parameter of **NewPropertyInterface()**) is not destroyed upon calling of this function. It is the API user responsibility to free it using **FreeValueInterface()**. Returns *false* if the deallocation failed.

Member Functions Documentation

label_c get_label () const

Gets the property's label.

metadata_list *get_dic_metadata_sets () const

Returns a list of all the possible metadata sets that can be used has a value of this property. If the property is not a set, the list will be empty. User is responsible for the deletion of the list returned.

const I_value *get_value () const

Gets the property's value.

bool free_metadata_sets (metadata_list *) const

Frees the list returned by `get_dic_metadata_sets()`. Returns *true* if successful.

6.4 I_value Class Reference

```
#include <I_value.hpp>
```

This class stores the value of a property (a memory pointer of a given type).

Public Member Functions

const uint8_t *get_data () const**size_t get_size () const****value_type get_type () const**

Constructor and Destructor Documentation

bool NewValueInterface (I_value **, value_type, size_t, const uint8_t *)

Retrieves a pointer on an **I_value** interface. The second parameter defines the MXF type of this value; the third one its size in bytes and finally the fourth one is a memory pointer on the data. Return *false* if the allocation failed.

bool FreeValueInterface(I_value **)

Frees an **I_value** interface. The data (*const uint8_t** parameter of **NewValueInterface()**) is not destroyed upon calling of this function. It is the API user responsibility to free it. Return *false* if the deallocation failed.

Member Functions Documentation

const uint8_t *get_data () const

Gets a memory pointer on the value's data. This pointer should be directly cast into the appropriate type depending on the return value of the function **get_type()**. Returned pointer must not be deleted.

size_t get_size () const

Gets the size in bytes of the data pointed by **get_data()**.

value_type get_type () const

Gets the MXF type of this value.

Related Documentation

enum value_type

<i>MXF_BOOLEAN</i>	int8_t[1]
<i>MXF_INT8</i>	int8_t[1]
<i>MXF_INT16</i>	int16_t[1]
<i>MXF_INT32</i>	int32_t[1]
<i>MXF_INT64</i>	int64_t[1]
<i>MXF_UINT8</i>	uint8_t[1]
<i>MXF_UINT16</i>	uint16_t[1]
<i>MXF_UINT32</i>	uint32_t[1]
<i>MXF_UINT64</i>	uint64_t[1] (also used with MXF types Position and Length [377M])
<i>MXF_UUID</i>	uint8_t[16] (also used with MXF type UL [377M])
<i>MXF_UMID32</i>	I_umid*
<i>MXF_UMID64</i>	I_umid64*
<i>MXF_RATIONAL</i>	uint32_t[2] (numerator / denominator)
<i>MXF_TIMESTAMP</i>	uint16_t[7] (year / month / day / hour / min / sec / millisec/4)
<i>MXF_ISO7_STRING</i>	char[n]
<i>MXF_UTF16_STRING</i>	wchar_t[n] (sizeof(wchar_t) = 2 on Linux, =4 on Windows).
<i>MXF_PRODUCT_VERSION</i>	uint16_t[5] (major / minor / patch / build / release)
<i>MXF_STRONG_REF</i>	I_metadata*
<i>MXF_WEAK_REF</i>	I_metadata*
<i>MXF_ARRAY_BOOLEAN</i>	int8_t[n] (ARRAY also used for MXF type Batch [377M])
<i>MXF_ARRAY_INT8</i>	int8_t[n]
<i>MXF_ARRAY_INT16</i>	int16_t[n]
<i>MXF_ARRAY_INT32</i>	int32_t[n]
<i>MXF_ARRAY_INT64</i>	int64_t[n]
<i>MXF_ARRAY_UINT8</i>	uint8_t[n]
<i>MXF_ARRAY_UINT16</i>	uint16_t[n]
<i>MXF_ARRAY_UINT32</i>	uint32_t[n]
<i>MXF_ARRAY_UINT64</i>	uint64_t[n]
<i>MXF_ARRAY_UUID</i>	uint16_t[16*n]
<i>MXF_ARRAY_STRONG_REF</i>	I_metadata*[n]
<i>MXF_ARRAY_WEAK_REF</i>	I_metadata*[n]
<i>MXF_ARRAY_UMID32</i>	I_umid*[n]
<i>MXF_ARRAY_UMID64</i>	I_umid64*[n]
<i>MXF_UNKNOWN</i>	type not recognized by MXFTk

This enumeration defines all the types available in an MXF file. The glossary [377M] describes them precisely. The list enumerated just above links the MXF type to its C++ *typedef* (the variable n defines a positive integer). The pointer returned by **get_data()** shall be cast into the appropriate type listed above. *MXF_STRONG_REF*, *MXF_WEAK_REF*, *MXF_ARRAY_STRONG_REF* and *MXF_ARRAY_WEAK_REF* types enable the construction of an **I_metadata** tree.

bool iso7_to_utf16 (wchar_t *, const char *, int)

bool utf16_to_iso7 (char *, const wchar_t *, int)

These functions convert an ISO7 string into a Unicode UTF16 string and vice versa. The integer states the number of characters in the string to be converted (not its length in bytes).

Warning: converting an UTF16 sting into an ISO7 string is not always possible.

Warning: On Linux platform, *wchar_t* type is four bytes long. Therefore only two bytes from each *wchar_t* will be effectively used when manipulating UTF16 string.

6.5 I_umid Class Reference

```
#include <I_umid.hpp>
```

This class describes a UMID identifying universally and uniquely a material. It may contain geographical and date information. It may also indicate who or what generated this material. MXFTk supplies this class to let the API user specifies its own values of UMID; however it can be simply ignored in which case MXFTk will generate default values.

Public Member Functions

```
const uint8_t *get_ptr () const  
const char *get_str () const
```

Constructor and Destructor Documentation

```
bool NewUmidInterfaceByArray (I_umid **, uint8_t *)
```

Retrieves a pointer on an **I_umid** interface. The second parameter contains the memory address of a 32-byte long buffer containing the UMID value. This buffer can be freed after calling this function. Returns *false* if the allocation failed.

```
bool NewUmidInterfaceByString (I_umid **, const char *)
```

Retrieves a pointer on an **I_umid** interface. The second parameter is the UMID value stored in a string. It represents 32 bytes in a hexadecimal form, each byte being separated by a dot. Returns *false* if the allocation failed.

```
bool FreeUmidInterface (I_umid **)
```

Frees an **I_umid** interface. Returns *false* if the deallocation failed.

Member Functions Documentation

```
const uint8_t *get_ptr () const
```

Returns the memory address of a 32-byte long buffer containing the UMID value. This buffer must not be edited or deleted.

```
const char *get_str () const
```

Returns a string containing the value of this UMID. It represents 32 bytes in a hexadecimal form, each byte being separated by a dot. Returned pointer must not be deleted.

6.6 I_umid64 Class Reference

```
#include <I_umid.hpp>
```

This class describes a 64-byte UMID universally and uniquely identifying a material. It may contain geographical and date information. It may also indicate who or what generated this material. MXFTk supplies this class to let the API user specifies its own values of UMID; however it can be simply ignored in which case MXFTk will generate default values. 64-byte UMID is used by the Descriptive Metadata Scheme 1 (DMS1).

Public Member Functions

```
const uint8_t *get_ptr () const  
const char *get_str () const
```


Constructors and Destructor Documentation

bool NewUmid64InterfaceByArray (I_umid64 **, uint8_t *)

Retrieves a pointer on an **I_umid64** interface. The second parameter contains the memory address of a 64-byte long buffer containing the UMID value. This buffer can be freed after calling this function. Returns *false* if the allocation failed.

bool NewUmid64InterfaceByString (I_umid64 **, const char *)

Retrieves a pointer on an **I_umid64** interface. The second parameter is the UMID value stored in a string. It represents 64 bytes in a hexadecimal form, each byte being separated by a dot. Returns *false* if the allocation failed.

bool FreeUmid64Interface (I_umid64 **)

Frees an **I_umid64** interface. Returns *false* if the deallocation failed.

Member Functions Documentation

const uint8_t *get_ptr () const

Returns the memory address of a 64-byte long buffer containing the UMID value. This buffer must not be edited or deleted.

const char *get_str () const

Returns a string containing the value of this UMID. It represents 64 bytes in a hexadecimal form, each byte being separated by a dot. Returned pointer must not be deleted.

7. Timecode Material

Timecode material allows the definition of the timecode used while playing an **I_generic_material** or an **I_concrete_material**. When added to an **I_generic_material** it corresponds to the file “play out” and must be continuous (no timecode redefinition while playing). When added to an **I_concrete_material** it corresponds to the file “play in” of the data embedded in the file and can be discontinuous. MXFTk generates default timecode materials so that the user does not necessarily need to define them. Only MXFTk Advanced version will let you redefine the timecode of your MXF files.

7.1 I_timecode_material Class Reference

```
#include <I_timecode_material.hpp>
```

An **I_timecode_material** is an interface representing a timecode track. Therefore, it is defined by an origin, a duration and an edit rate. Timecode material schematizes the timecode in use while playing the corresponding generic material.

```
opencube::MXFTK_NAMESPACE::I_generic_material
```



```
opencube::MXFTK_NAMESPACE::I_timecode_material
```

Public Member Functions

```
bool append_timecode_component (I_timecode_component *)
```

```
bool free_timecode_list (ordered_timecode_list *)
```

```
rational *get_edit_rate ()
```

```
int64_t get_origin () const
```

```
ordered_timecode_list *get_timecode_list ()
```

```
int64_t get_total_duration () const
```

Constructor and Destructor Documentation

```
bool NewTimecodeMaterialInterface (I_timecode_material **, int64_t, rational *)
```

Retrieves a pointer on an **I_timecode_material** interface. The second parameter defines the starting edit unit (origin) of the timecode track created. The third parameter specifies its edit rate while the duration will be automatically computed as **I_timecode_component** will be added to this material (**append_timecode_component()**). Returns *false* if the allocation failed.

```
bool FreeTimecodeMaterialInterface (I_timecode_material **)
```

Frees an **I_timecode_material** interface. Returns *false* if the deallocation failed.

Member Functions Documentation

```
void append_timecode_component (I_timecode_component *)
```

Adds to the end of the timecode track a new timecode definition. In most cases, a unique **I_timecode_material** should be embedded in an **I_timecode_material**. This ensures continuity while playing the audiovisual material (requirement of MXF file play out). Timecode redefinition (adding several **I_timecode_component**) lets you create discontinuous timelines. You may set the duration of the **I_timecode_component** to -1 in order to leave MXFTk compute it automatically whenever possible.

```
void free_timecode_list (ordered_timecode_list *)
```

Frees the list returned by **get_timecode_list()**.

rational *get_edit_rate ()

Gets the edit rate of the underlying timecode track.

Warning: the edit rate of a timecode track is not necessarily equal to the frame rate of its timecode components.

int64_t get_origin () const

Gets the origin of the timecode track measured in edit units.

ordered_timecode_list *get_timecode_list ()

Retrieves the ordered list of **I_timecode_component** from this timecode track. A timecode component states a starting timecode and duration. Therefore, adjacent **I_timecode_component** define an **I_timecode_material** just as adjacent **I_track_item** define an **I_track**.

int64_t get_total_duration () const

Gets the timecode track duration measured in edit units.

7.2 I_timecode Class Reference

```
#include <I_timecode.hpp>
```

This class represents a timecode value; in other words a time instance expressed in the following format "hour:minute:second:frame". This structure is used to create timecode components of an **I_timecode_material** or to access audiovisual data at a given timecode.

Public Member Functions

```
bool add (uint64_t)
bool equal_to (I_timecode *) const
bool get_drop_frame () const
uint8_t get_frame () const
uint64_t get_frame_count ()
uint16_t get_frame_rate () const
uint8_t get_hour () const
uint8_t get_minute () const
uint8_t get_second () const
const char * get_time () const
bool greater_than (I_timecode *) const
bool next ()
bool previous ()
bool smaller_than (I_timecode *) const
bool sub (uint64_t)
```

Constructor and Destructor Documentation

bool CopyTimecodeInterface (I_timecode **, const I_timecode *)

Copies an **I_timecode** interface. Returns *false* if the copy failed.

bool NewTimecodeInterface (I_timecode **, const char *, uint16_t, bool)

Retrieves a pointer on an **I_timecode** interface. The second parameter is a string defining the timecode value according to the following rule: "XX:XX:XX:XX" ("05:14:00:08" for instance). The third parameter specifies the frame rate of this timecode and the last Boolean value states whether it is a drop frame timecode (drop frame timecode is only allowed with 25, 30 or 60 frames per second). Returns *false* if the allocation failed.

bool NewTimecodeInterfaceByValue (I_timecode **, uint16_t, uint16_t, uint16_t, uint16_t, uint16_t, bool)

Retrieves a pointer on an **I_timecode** interface. *uint16_t* parameters are respectively the hour, minute, second, frame and frame rate of this timecode. The last Boolean value states whether it is a drop frame timecode (drop frame timecode is only allowed with 25, 30 or 60 frames per second). Returns *false* if the allocation failed.

bool FreeTimecodeInterface(I_timecode **)

Frees an **I_timecode** interface. Returns *false* if the deallocation failed.

Member Functions Documentation

bool add (uint64_t)

Adds a given number of frames to the current timecode value. Returns *false* if an error occurred.

bool equal_to (I_timecode *) const

Returns *true* if the current timecode is equal to the timecode specified, *false* otherwise.

bool get_drop_frame () const

State whether a drop frame timecode is in use.

uint8_t get_frame () const

Returns the frame digits of this timecode. If timecode is 05:25:13:24 it will return 24.

uint64_t get_frame_count () const

Returns the timecode value as a number of frames elapsed from timecode 00:00:00:00. For instance, if the timecode value is 00:02:10:14 and the frame rate is 25, the returned value will be $2*60*25+10*25+14+1 = 3265$.

uint16_t get_frame_rate () const

Get the frame rate of this timecode.

uint8_t get_hour () const

Returns the hour digits of this timecode. If timecode is 05:25:13:24 it will return 5.

uint8_t get_minute () const

Returns the minute digits of this timecode. If timecode is 05:25:13:24 it will return 25.

uint8_t get_second () const

Return the second digits of this timecode. If timecode is 05:25:13:24 it will return 13.

const char *get_time () const

Gets the current timecode value as a string. Returned pointer must not be deleted.

bool greater_than (I_timecode *) const

Returns *true* if the current timecode is greater than the timecode specified, *false* otherwise.

bool next ()

Increases current timecode.

bool previous ()

Decreases current timecode.

bool smaller_than (I_timecode *) const

Returns *true* if the current timecode is smaller than the timecode specified, *false* otherwise.

void sub (int64_t)

Subtracts a given number of frames to the current timecode value.

8. Streaming

The following classes should be used when manipulating MXF files in a streaming environment (IEEE 1394 port, videotape recorder, etc.). MXFTk lets you stream the input audiovisual data, the descriptive metadata and the MXF files being generated or read. When wrapping an MXF file you may even receive the audiovisual data and the descriptive metadata from several streams while outputting the MXF stream on the fly. MXFTk user is strongly invited to refer to the examples “active_stream_wrapper”, “passive_stream_wrapper”, “mxf_stream_unwrapper” and “opla_stream_wrapper”. It is also best to feel comfortable with MXF file wrapping/unwrapping in a non-streaming environment before going through this chapter.

All the following classes are pure virtual classes that the user must implement. They work as a set of functions that MXFTk will call whenever he needs some information or data from the corresponding streaming device. Therefore, MXFTk user is entitled to implement these classes to feed MXFTk with data whenever it needs to.

8.1 I_essence_stream_task Class Reference

```
#include <streaming.hpp>
```

This class should be used when the audiovisual data used to build an **I_concrete_material** is originating from a streaming device (for instance when receiving the stream of a video tape and wrapping it on-the-fly in an MXF file). MXFTk user is entitled to derive this class in order to provide an implementation for the streaming device.

Public Member Functions

```
bool get_active_blocking_mode (unsigned int)
size_t data_request (unsigned int, uint8_t *, size_t)
size_t get_buffer_size (unsigned int)
bool get_passive_mode ()
void get_stream_info (unsigned int, essence_source &, bool &, unsigned int &)
void get_mpeg_es_info (unsigned int, unsigned int, bool &)
unsigned int get_total_number_of_streams ()
void get_audio_mxf_file_descriptor (unsigned int, I_mxf_file_descriptor *&descriptor)
void get_video_mxf_file_descriptor (unsigned int, I_mxf_file_descriptor *&descriptor)
uint64_t stream_size (unsigned int)
int64_t duration (unsigned int)
```

Member Functions Documentation

bool get_active_blocking_mode (unsigned int)

This function is called before launching the streaming process. It should return *true* if you choose to perform the streaming task for the designed track (unsigned int parameter) in the blocking mode. This mode will be effective only if you also set the active streaming mode thanks to the function **get_passive_mode()**.

- **BLOCKING MODE:** MXFTk performs no copy of the buffer sent by the user. User should call the function **I_concrete_material::write()** in order to feed MXFTk with data. It will return from the function only once the data has been entirely written. Because there is no buffering, the data must be sent in the exact way it will be wrapped. This means the buffers should be sent frame by frame, one after the other in the order they will be wrapped. The user may use the function **I_mxf_file::waiting_stream()** in order to have a guideline to figure out the order in which it should send the data.
- **BUFFERING MODE:** The data sent to MXFTk is immediately copied and bufferized for later processing. This mode is more convenient to use for the user has it does not need to care about the order and size of buffers he sends to MXFTk. However it may be less efficient as it requires extra copies of buffer.

size_t data_request (unsigned int, uint8_t *, size_t)=0

This function is called whenever MXFTk requests data from the stream. The first parameter is the stream identifier (comprised between 0 and **get_total_number_of_streams()-1**). It is used to keep a reference on the stream during

successive calls to **data_request()**. The second parameter is the buffer where the data sent to MXFTk should be written. The third parameter indicates how many bytes are required (and hence ideally it is also the size of the buffer passed (second parameter)). Finally the value returned should indicate how many bytes have been written in the buffer. Users of previous MXFTk versions should be careful as it is no longer required to call the **I_concrete_material::write()** function to feed MXFTk.

This function will be called regularly upon flushing of the MXF file to notify the user that the flushing process is in wait of data from the streaming device. A correct implementation should stop the current process using a mutex as long as the requested data is not available from the stream. Once ready, the data should be written in the buffer provided. Finally, if no more data is to be received then this should also be notified by returning 0 from the function.

size_t get_buffer_size (unsigned int)=0

This function is called before launching the streaming process. It gives the opportunity to set the size of MXFTk's internal buffer used to store the data received from the stream. You may define a different size for each stream id. Depending on your system, changing the buffer's size may help to improve your performances. It is always better to increase the size of the buffers in order to handle throughput variations. This function will be called only once for each stream id, the size cannot be changed during the streaming process.

bool get_passive_mode ()=0

This function is called before launching the streaming process. It should return *true* if you choose to perform the streaming task in passive mode or *false* if you choose active mode:

- **PASSIVE MODE:** MXFTk calls the function **data_request()** whenever it requires some data. You must provide an implementation of the function **data_request()** in that case and you should perform the **I_concrete_material::write()** calls only within this function. MXFTk controls the wrapping and you may send data only when it notifies you to do so.
- **ACTIVE MODE:** You can send data to MXFTk whenever you want still using the function **I_concrete_material::write()**. You do not need to implement the function **data_request()** in that case and you must send the data after calling the function **I_mxf_file::flush()**.

bool get_stream_info (unsigned int, essence_source &, bool &, unsigned int &)

You must implement this function. For performance reasons, MXFTk needs you to specify the nature of the stream you will send. This function lets you do so. The first parameter is the stream id. The second parameter defines the type of the essence for this stream id (DV, MPEG, etc... refer to **I_essence_type** for a complete list of possible types). The third parameter indicates if it is a video or audio stream. However in the case of a DV stream it is used to know if the audio from the DV should appear as a track in the MXF file. If set to true, the audio from the DV will be ignored. Finally the last parameter is only used when the input stream is a MPEG Program or Transport Stream and is used to indicate how many MPEG Elementary Streams they embed.

bool get_mpeg_es_info (unsigned int, unsigned int, bool &)

You must implement this function if one of the input streams is an MPEG Program Stream or Transport Stream. The first parameter identifies the stream and the second one identifies the "elementary streams" from the MPEG PS or TS. Its value ranges from 0 to the number of sub-streams set in **get_stream_info**. There is no relation between this id and the id of the MPEG ES stored in the MPEG stream. The Boolean should be set to *true* if the corresponding sub-stream is a video stream or set to *false* if it is an audio one.

unsigned int get_total_number_of_streams ()

This function is called before launching the streaming process. It is used to set the total number of streams that will be embedded on the **I_concrete_material** being created (just as several files may be required to build a concrete material).

void get_audio_mxf_file_descriptor (unsigned int, I_mxf_file_descriptor *&descriptor)

This function is called before launching the streaming process. The user may use this function in order to build the audio essence descriptor corresponding to the stream id provided (first parameter). Most of the time, you will not need to implement this function as MXFTk is capable to build descriptors by analyzing the content of the data you will send. However when wrapping essences from which descriptors cannot be constructed directly from the essence content (Uncompressed raw files, PCM data without header, etc...) user will need to build the descriptors

to create a valid MXF file.

void get_video_mxf_file_descriptor (unsigned int, I_mxf_file_descriptor *&descriptor)

This function is called before launching the streaming process. The user may use this function in order to build the video essence descriptor corresponding to the stream id provided (first parameter). Most of the time, you will not need to implement this function as MXFTk is capable to build descriptors by analyzing the content of the data you will send. However when wrapping essences from which descriptors cannot be constructed directly from the essence content (Uncompressed raw files, PCM data without header, etc...) user will need to build the descriptors to create a valid MXF file.

uint64_t stream_size (unsigned int)

This function can be implemented in order to set the total size in bytes of the stream that will be wrapped.

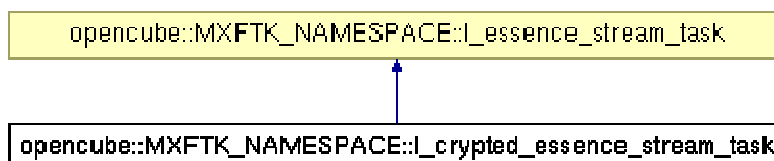
int64_t duration (unsigned int)

This function can be implemented in order to set the total duration in edit units of the stream that will be wrapped.

8.2 I_crypted_essence_stream_task Class Reference

```
#include <streaming.hpp>
```

This class should be used when the audiovisual data used to build an **I_concrete_material** is originating from a streaming device (for instance when receiving the stream of a video tape and wrapping it on-the-fly in an MXF file) and should be crypted. MXFTk user is entitled to derive this class in order to provide an implementation for his streaming device.



Public Member Functions

```
const uint8_t * get_cipher_key (unsigned int)
const uint8_t * get_cryptographic_key_id (unsigned int)
bool do_crypt (unsigned int)
uint32_t get_plaintext_offset (unsigned int)
```

Member Functions Documentation

const uint8_t * get_cipher_key (unsigned int)

Implement this function so that it returns the cipher key that will be used to encrypt the stream with the corresponding stream id (first parameter).

const uint8_t * get_cryptographic_key (unsigned int)

Implement this function so that it returns the cryptographic key that will be used to encrypt the stream with the corresponding stream id (first parameter).

bool do_crypt (unsigned int)

Implement this function so that it returns *true* if the stream with the corresponding stream id (parameter) must be encrypted.

uint32_t get_plaintext_offset (unsigned int)

Implement this function so that it returns the plaintext offset that will be used to encrypt the stream with the

corresponding stream id (first parameter). The plaintext offset is the offset to the encrypted data within a frame.

8.3 I_input_metadata_stream_task Class Reference

```
#include <streaming.hpp>
```

This class should be used when the descriptive metadata to be embedded on an MXF file is originating from a streaming device. MXFTk user is entitled to derive this class in order to provide an implementation for the corresponding streaming device.

Public Member Functions

```
void update (I_generic_material *, I_track *, int64_t)
```

Member Functions Documentation

virtual void update (I_generic_material *, I_track *, int64_t)=0

This function is called by MXFTk whenever some descriptive metadata can be written in the MXF file being created. It is called when a new partition is created. Therefore, it is mandatory to understand that the “key” parameters to use efficiently this function are **HEADER_REPETITION** and **PREFERRED_PARTITION_DURATION** that can be set thanks to the function **I_mxf_file::set_parameter()**. The larger the partition size, the less often this function **update()** will be called. On the contrary, the smaller the partition size, the more often the user will have the opportunity to update the file. However, increasing the number of partitions in your file may severely impact the overall performances.

The first and second parameter are respectively the **I_generic_material** and the **I_track** (metadata track) to be updated; while the **int64_t** is the current position (in edit units) on this descriptive metadata track. It is perfectly valid not to add metadata or update the descriptive metadata and simply returning from this function.

8.4 I_output_mxf_stream_task Class Reference

```
#include <streaming.hpp>
```

This class should be used when the MXF file being created needs to be retrieved on-the-fly. This can be the case, when sending the MXF stream to an MXF videotape recorder. MXFTk user is entitled to derive this class in order to provide an implementation for the corresponding streaming device. Calls to this class will be performed only during the **I_mxf_file::flush()** process.

Public Member Functions

```
void data_to_read (const uint8_t *, size_t)
```

```
bool seekable ()
```

```
uint64_t seekpos_request (uint64_t)
```

Member Functions Documentation

void data_to_read (const uint8_t *, size_t)

This function is called by MXFTk to notify that its internal output buffer is filled and that the MXF data is ready to be sent to the streaming device. The first parameter is a pointer on the MXF buffer and the second one is the number of bytes to be read from this buffer. MXFTk user should not free this buffer! After exiting this function, MXFTk always assume that all the data was read. Therefore, if your streaming device is not ready to receive this data you should make use of a mutex to stop the current process as long as necessary. In order to notify the end of the MXF stream and that no more data will be sent, the API will call this function with a NULL buffer.

bool seekable ()

This function is called by MXFTk to know if it will be allowed to seek in the output MXF stream. When wrapping an OpZero MXF file, it is mandatory to set the stream as seekable or the creation of the file will be impossible. Other types of MXF files do not necessarily require a seekable stream but allowing seek operations can help to produce closed and complete MXF files.

uint64_t seekpos_request (uint64_t)

Implement this function if you defined the stream as seekable. The parameter is the absolute position where the seek operation should be performed and the returned value should contain the position that was effectively reached.

8.5 I_input_mxf_stream_task Class Reference

```
#include <streaming.hpp>
```

This class should be used when the MXF file is received from a streaming device. This can be the case, when receiving the output signal of an MXF videotape player. MXFTk user is entitled to derive this class in order to provide an implementation for the corresponding streaming device. Calls to this class will be performed only during the **I_mxf_file** creation process. It also provides interfaces for seeking data to a given timecode on the streaming device. Due to their internal construction, some MXF files are not truly streamable. This would be the case when the video and audio data meant to be played together are both clip wrapped in the file. Because the reading of the file will be performed linearly, MXFTk will first read all of the video before reaching the audio. When building files to be read in a streaming environment, it is always best to avoid high operational patterns and to perform a frame wrapping of all the audiovisual data (the sources will be multiplexed so that they can be played simultaneously). You may also use this class to perform the update of an MXF file read from a streaming device.

Public Member Functions

```
void data_to_read (I_generic_material *, I_track *, I_source_clip *, uint64_t)
void data_request (I_mxf_file *, size_t)
size_t get_buffer_size ()
uint64_t seekpos_request (uint64_t)
I_timecode *seek_timecode (I_generic_material *, bool&)
bool seekable ()
void last_partition_offset (uint64_t)
uint64_t stream_size ()
void updated_data (const uint8_t *, size_t)
bool updated_mxf_file (I_mxf_file *, bool closed, bool complete, uint64_t mxf_stream_pos)
```

Member Functions Documentation

void data_to_read (I_generic_material *, I_track *, I_source_clip *, uint64_t)

This function is called by MXFTk to notify that some data is ready to be read on the **I_source_clip** from the given **I_track** in the given **I_generic_material**. The **uint64_t** parameter indicates the number of bytes to be read (size of the edit unit to be read). The data can be read with a single call to the function **I_source_clip::read_eu**. Seek functions of the source clip should not be used. If your system is not ready to read this data, the current process should be stopped as long as necessary.

void data_request (I_mxf_file *, size_t)

This function is called whenever MXFTk requests data from the MXF stream in order to continue the unwrapping process. The **size_t** parameter states how many bytes the internal buffer is waiting for. Calls to **I_mxf_file::write** should be performed once the data is ready. If the streaming device is not ready, then the current process should be stopped as long as necessary. When no more data is to be received, an explicit call to **I_mxf_file::end_of_stream** must be performed.

size_t get_buffer_size ()

This function is called before launching the streaming process. It gives the opportunity to set the size of MXFTk's internal buffer used to store the data received from the MXF stream. Depending on your system, changing the buffer's size may help to improve your performances. This function will be called only once, the size cannot be changed during the streaming process.

uint64_t seekpos_request (uint64_t)

User should implement this function only if the MXF streaming device has the capability to seek. During a linear streaming process, MXFTk will not call this function. This will occur only following an explicit seeking request from the user (thanks to the function **seek_timecode()**). The **uint64_t** parameter gives the absolute position where the streaming device should seek. Returned value should be the requested position. If you choose not to implement this function, **seek_timecode()** should always return NULL.

I_timecode *seek_timecode (I_generic_material *, bool&)

This function will be regularly called by MXFTk to give the opportunity to continue the MXF unwrapping process from a user-defined timecode. The streaming device should have the capability to seek and if the user chooses to implement this function, **seekpos_request()** must be implemented as well. The function should return the timecode to reach (relative to the timecode material of the parameter **I_generic_material**) or NULL if the unwrapping process should continue linearly from the current timecode. It is not possible to seek a timecode that has not been reached so far. You can set the value of the second parameter to *true*; this will force the streaming process to stop immediately.

bool seekable ()

You should implement this function to let MXFTk know if it is allowed to perform seek operations on this stream. Allowing seek operations may speed up the decoding process. If the function returns *true*, then you must provide your implementation of **seekpos_request()** and **stream_size()**. The return value of this function cannot be changed during the MXF decoding process. Note that the update of an MXF file in a streaming environment can only be performed if the stream is seekable. Hence in that case, this function should return *true* and you must implement **seekpos_request()**.

void last_partition_offset (uint64_t)

MXFTk calls this function whenever a new header metadata is being decoded. The parameter specifies the offset in the stream to reach the beginning of the partition containing this header metadata.

uint64_t stream_size ()

This function will be called by MXFTk to retrieve the total size of the stream. If this value cannot be evaluated, the function should always return 0. Specifying the size of the stream may speed up the decoding process. You must return the correct size of the stream if you set it to be seekable.

void updated_data (const uint8_t *, size_t)

You need to implement this function only if you are performing an update of the MXF file. MXFTk will call this function regularly while updating the file. It sends you a buffer and its size to be written at the current location of the stream.

bool updated_mxf_file (I_mxf_file *, bool closed, bool complete, uint64_t mxf_stream_pos)

The structure of an MXF file being streamed is likely to be changed during the decoding process. Several instances of the header metadata containing both the structural and descriptive metadata can be found in an MXF file. This is usually done to reflect the changes that occurred on the file while it was recorded. Therefore, the truthful header metadata is generally found at the end of the file. However, during a linear streaming process, this one cannot be reached before processing all the data; therefore MXFTk will regularly send updated version of the header metadata whenever a new one is detected. This lets the API user start the decoding process although part of the information is missing. When MXFTk calls this function, previous **I_mxf_file** should be considered as outdated and previous references to objects or interfaces should be ignored.

However, to help the user two Boolean values indicate if the **I_mxf_file** structure is still likely to change. If *closed*,

then the overall structure will not change (no more tracks or materials will be added or removed for instance) but some values are probably still missing (this is notably the case for the duration of the tracks). If the file is closed and complete, no more updates will occur, the **I_mxf_file** is entirely valid.

The **uint64_t** gives the number of bytes read from the MXF streaming device so far.

The function should return *true* if you wish to stop the decoding process. This is useful if you are not interested in the essence and want to stop the process as soon as you found a valid header metadata.

8.6 I_input_partial_mxf_stream_task Class Reference

```
#include <streaming.hpp>
```

This class should be used when the MXF file to partial restore is received from a streaming device. This can be the case, when receiving the output signal of an MXF videotape player. MXFTk user is entitled to derive this class in order to provide an implementation for the corresponding streaming device. Calls to this class will be performed only during the **I_mxf_file** partial restore process.

Public Member Functions

```
void data_request (I_mxf_file *, size_t)
size_t get_buffer_size ()
uint64_t seekpos_request (uint64_t)
bool seekable ()
uint64_t stream_size ()
```

Member Functions Documentation

void data_request (I_mxf_file *, size_t)

This function is called whenever MXFTk requests data from the MXF stream in order to continue the partial restore process. The *size_t* parameter states how many bytes the internal buffer is waiting for. Calls to **I_mxf_file::write** should be performed once the data is ready. If the streaming device is not ready, then the current process should be stopped as long as necessary. When no more data is to be received, an explicit call to **I_mxf_file::end_of_stream** must be performed.

size_t get_buffer_size ()

This function is called before launching the streaming process. It gives the opportunity to set the size of MXFTk's internal buffer used to store the data received from the MXF stream. Depending on your system, changing the buffer's size may help to improve your performances. This function will be called only once, the size cannot be changed during the streaming process.

uint64_t seekpos_request (uint64_t)

User should implement this function only if the MXF streaming device has the capability to seek. The *uint64_t* parameter gives the absolute position where the streaming device should seek. Returned value should be the requested position.

bool seekable ()

You should implement this function to let MXFTk knows if it is allowed to perform seek operations on this stream. Allowing seek operations is likely to speed up the partial restore process. If the function returns *true*, then you must provide your implementation of **seekpos_request()** and **stream_size()**. The return value of this function cannot be changed during the MXF decoding process.

uint64_t stream_size ()

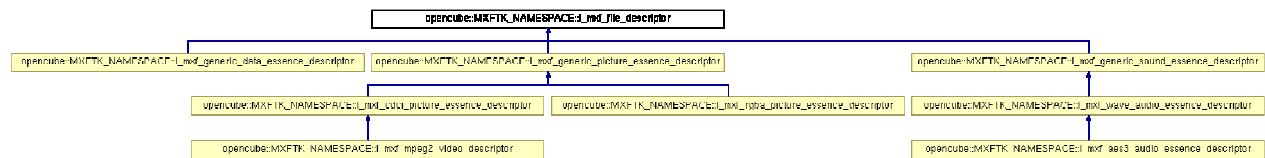
This function will be called by MXFTk to retrieve the total size of the stream. If this value cannot be evaluated, the function should always return 0. Specifying the size of the stream is likely to speed up the partial restore process. You must return the correct size of the stream if you set it to be seekable.

9. Essence Descriptors

The following classes can be used to create the complete essence descriptors. Most of the time, you will not need to implement this function because MXFTk can build them automatically by analyzing the source files to be wrapped. However, in certain cases, this information can not be retrieved from the source stream. In that case, the user should build these classes in order to create a valid MXF file. In this manual we will not go through the definitions of all the properties set in the descriptors but we invite the reader to refer to the corresponding norms for a complete reference.

9.1 I_mxf_file_descriptor Class Reference

```
#include <I_mxf_file_descriptor.hpp>
```



Public Member Functions

```
void set_sample_rate (const uint32_t numerator, const uint32_t denominator)
```

```
void set_container_duration (const uint64_t)
```

```
void set_essence_container_ul (const char *)
```

```
void set_codec_ul (const char *)
```

```
void add_subdescriptor (const I_mxf_subdescriptor *)
```

Constructor and Destructor Documentation

```
bool NewMxfileDescriptorInterface (I_mxf_file_descriptor **)
```

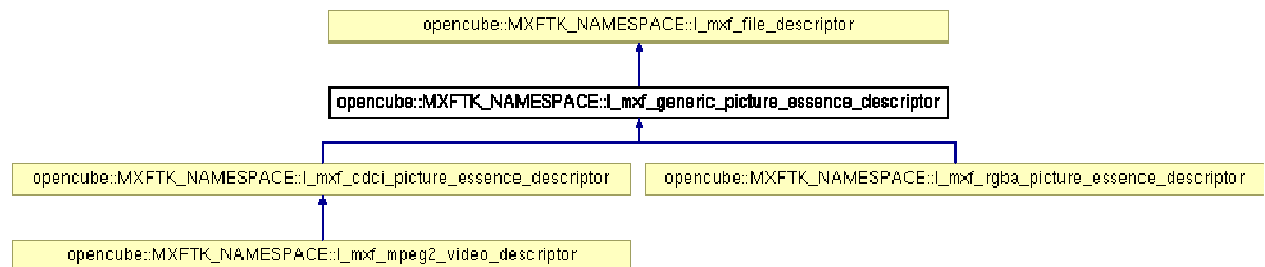
Retrieves a pointer on an **I_mxf_file_descriptor** interface. Returns *false* if the allocation failed.

```
bool FreeMxfileDescriptorInterface (I_mxf_file_descriptor **)
```

Frees an **I_mxf_file_descriptor** interface. Returns *true* if successful.

9.2 I_mxf_generic_picture_essence_descriptor Class Reference

```
#include <I_mxf_file_descriptor.hpp>
```



Public Member Functions

```
void set_signal_standard (const uint8_t)
```

```
void set_frame_layout (const uint8_t)
```



```

void set_stored_width (const uint32_t)
void set_stored_height (const uint32_t)
void set_stored_f2_offset (const int32_t)
void set_sampled_width (const uint32_t)
void set_sampled_height (const uint32_t)
void set_sampled_x_offset (const int32_t)
void set_sampled_y_offset (const int32_t)
void set_display_width (const uint32_t)
void set_display_height (const uint32_t)
void set_display_x_offset (const int32_t)
void set_display_y_offset (const int32_t)
void set_display_f2_offset (const int32_t)
void set_aspect_ratio (const uint32_t, const uint32_t)
void set_active_format_descriptor (const uint8_t)
void set_video_line_map (const int32_t, const int32_t)
void set_alpha_transparency (const uint8_t)
void set_capture_gamma_ul (const char *)
void set_image_alignment_offset (const uint32_t)
void set_image_start_offset (const uint32_t)
void set_image_end_offset (const uint32_t)
void set_field_dominance (const uint8_t)
void set_picture_essence_coding_ul (const char *)

```

Constructor and Destructor Documentation

bool NewMxfGenericPictureEssenceDescriptorInterface
(**I_mxf_generic_picture_essence_descriptor ****)

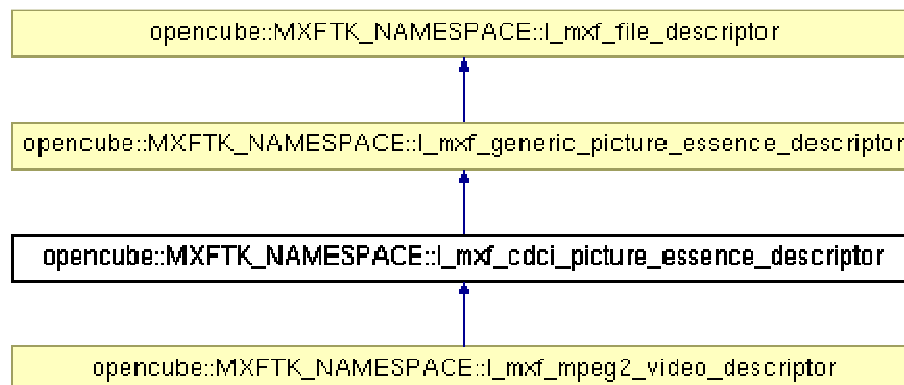
Retrieves a pointer on an **I_mxf_generic_picture_essence_descriptor** interface. Returns *false* if the allocation failed.

bool FreeMxfGenericPictureEssenceDescriptorInterface
(**I_mxf_generic_picture_essence_descriptor ****)

Frees an **I_mxf_generic_picture_essence_descriptor** interface. Returns *true* if successful.

9.3 I_mxf_cdc_picture_essence_descriptor Class Reference

```
#include <I_mxf_file_descriptor.hpp>
```



Public Member Functions

```
void set_component_depth (const uint32_t)
void set_horizontal_subsampling (const uint32_t)
void set_vertical_subsampling (const uint32_t)
void set_color_siting (const uint8_t)
void set_reversed_byte_order (const bool)
void set_padding_bits (const int16_t)
void set_alpha_sample_depth (const uint32_t)
void set_black_ref_level (const uint32_t)
void set_white_ref_level (const uint32_t)
void set_color_range (const uint32_t)
```

Constructor and Destructor Documentation

bool NewMxfCDCIPictureEssenceDescriptorInterface (I_mxf_cdc_picture_essence_descriptor **)

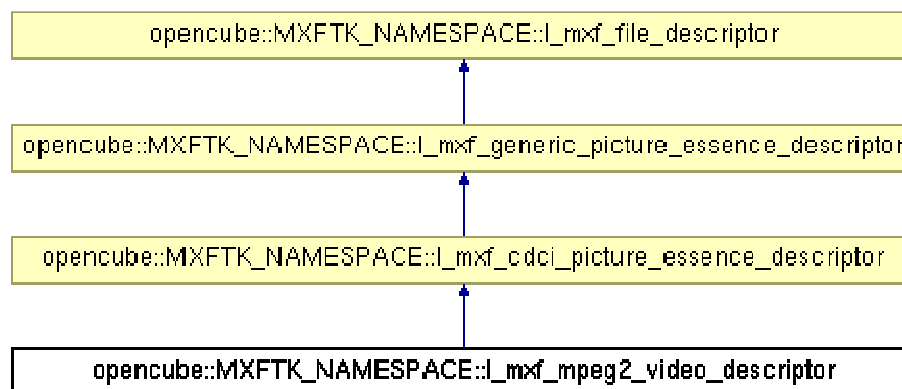
Retrieves a pointer on an `I_mxf_cdc_picture_essence_descriptor` interface. Returns *false* if the allocation failed.

bool FreeMxfCDCIPictureEssenceDescriptorInterface (I_mxf_cdc_picture_essence_descriptor **)

Frees an `I_mxf_cdc_picture_essence_descriptor` interface. Returns *true* if successful.

9.4 I_mxf_mpeg2_video_descriptor Class Reference

```
#include <I_mxf_file_descriptor.hpp>
```



Public Member Functions

```
void set_single_sequence (const bool)
void set_constant_b_frames (const bool)
void set_coded_content_type (const uint8_t)
void set_low_delay (const bool)
void set_closed_gop (const bool)
void set_identical_gop (const bool)
void set_max_gop (const uint16_t)
void set_b_picture_count (const uint16_t)
void set_bit_rate (const uint32_t)
void set_profile_and_level (const uint8_t)
```

Constructor and Destructor Documentation

bool NewMxfMpeg2VideoDescriptorInterface (I_mxf_mpeg2_video_descriptor **)

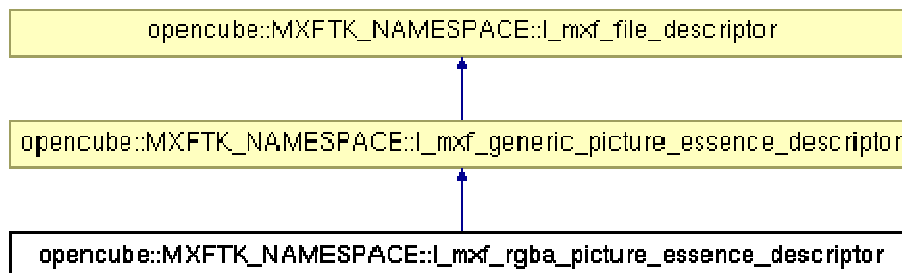
Retrieves a pointer on an **I_mxf_mpeg2_video_descriptor** interface. Returns *false* if the allocation failed.

bool FreeMxfMpeg2VideoDescriptorInterface (I_mxf_mpeg2_video_descriptor **)

Frees an **I_mxf_mpeg2_video_descriptor** interface. Returns *true* if successful.

9.5 I_mxf_rgba_picture_essence_descriptor Class Reference

#include <I_mxf_file_descriptor.hpp>



Public Member Functions

void set_component_max_ref (const uint32_t)
void set_component_min_ref (const uint32_t)
void set_alpha_max_ref (const uint32_t)
void set_alpha_min_ref (const uint32_t)
void set_scanning_direction (const uint8_t)
void set_pixel_layout (const uint8_t *, const uint32_t)
void set_palette (const uint8_t *, const uint32_t)
void set_palette_layout (const uint8_t *, const uint32_t)

Constructor and Destructor Documentation

bool NewMxfRGBAPictureEssenceDescriptorInterface (I_mxf_rgba_picture_essence_descriptor **)

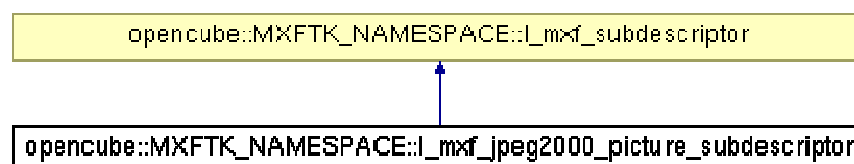
Retrieves a pointer on an **I_mxf_rgba_picture_essence_descriptor** interface. Returns *false* if the allocation failed.

bool FreeMxfRGBAPictureEssenceDescriptorInterface (I_mxf_rgba_picture_essence_descriptor **)

Frees an **I_mxf_rgba_picture_essence_descriptor** interface. Returns *true* if successful.

9.6 I_mxf_jpeg2000_picture_subdescriptor Class Reference

#include <I_mxf_file_descriptor.hpp>



Public Member Functions

```

void set_r_siz (const uint16_t)
void set_x_siz (const uint32_t)
void set_y_siz (const uint32_t)
void set_xo_siz (const uint32_t)
void set_yo_siz (const uint32_t)
void set_xt_siz (const uint32_t)
void set_yt_siz (const uint32_t)
void set_xto_siz (const uint32_t)
void set_yto_siz (const uint32_t)
void set_c_siz (const uint16_t)
void add_picture_component_sizing (const uint8_t, const uint8_t, const uint8_t)
void set_coding_style_default (const uint8_t *, const uint32_t)
void set_quantization_default (const uint8_t *, const uint32_t)

```

Constructor and Destructor Documentation

```

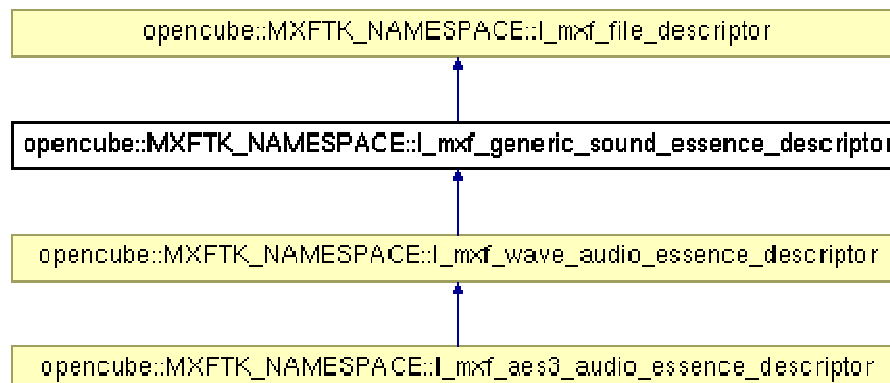
bool NewMxfJpeg2000PictureSubdescriptorInterface (I_mxf_jpeg2000_picture_subdescriptor **)
    Retrieves a pointer on an I_mxf_jpeg2000_picture_subdescriptor interface. Returns false if the allocation failed.

bool FreeMxfJpeg2000PictureSubdescriptorInterface (I_mxf_jpeg2000_picture_subdescriptor **)
    Frees an I_mxf_jpeg2000_picture_subdescriptor interface. Returns true if successful.

```

9.7 I_mxf_generic_sound_essence_descriptor Class Reference

```
#include <I_mxf_file_descriptor.hpp>
```



Public Member Functions

```

void set_audio_sampling_rate (const uint32_t, const uint32_t)
void set_locked_unlocked (const bool)
void set_audio_ref_level (const int8_t)
void set_electro_spatial_formulation (const uint8_t)
void set_channel_count (const uint32_t)
void set_quantization_bits (const uint32_t)
void set_dial_norm (const uint32_t)

```



```
void set_sound_essence_compression_ul (const char *)
```

Constructor and Destructor Documentation

bool NewMxfGenericSoundEssenceDescriptorInterface

(I_mxf_generic_sound_essence_descriptor**)

Retrieves a pointer on an **I_mxf_generic_sound_essence_descriptor** interface. Returns *false* if the allocation failed.

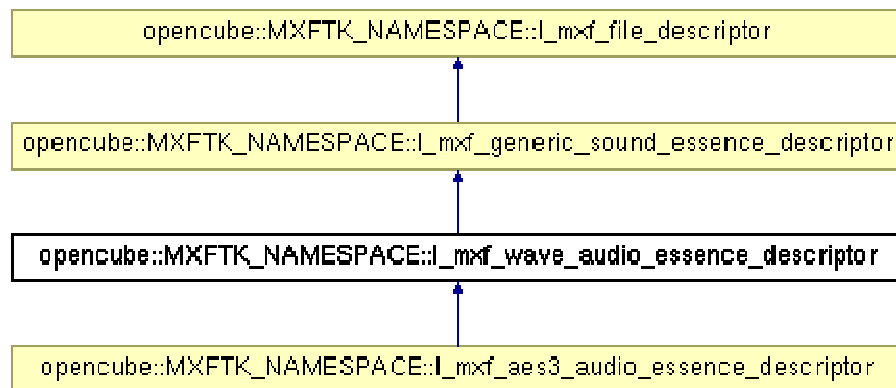
bool FreeMxfGenericSoundEssenceDescriptorInterface

(I_mxf_generic_sound_essence_descriptor **)

Frees an **I_mxf_generic_sound_essence_descriptor** interface. Returns *true* if successful.

9.8 I_mxf_wave_audio_essence_descriptor Class Reference

```
#include <I_mxf_file_descriptor.hpp>
```



Public Member Functions

```

void set_block_align (const uint16_t)
void set_sequence_offset (const uint8_t)
void set_avg_bps (const uint32_t)
void set_channel_assignment_ul (const char *)
void set_peak_envelope_version (const uint32_t)
void set_peak_envelope_format (const uint32_t)
void set_points_per_peak_value (const uint32_t)
void set_peak_envelope_block_size (const uint32_t)
void set_peak_channels (const uint32_t)
void set_peak_frames (const uint32_t)
void set_peak_of_peaks_position (const uint64_t)
void set_peak_envelope_timestamp (const uint64_t)
void set_peak_envelope_data (const uint8_t *, const uint32_t)

```

Constructor and Destructor Documentation

bool NewMxfWaveAudioEssenceDescriptorInterface (I_mxf_wave_audio_essence_descriptor **)

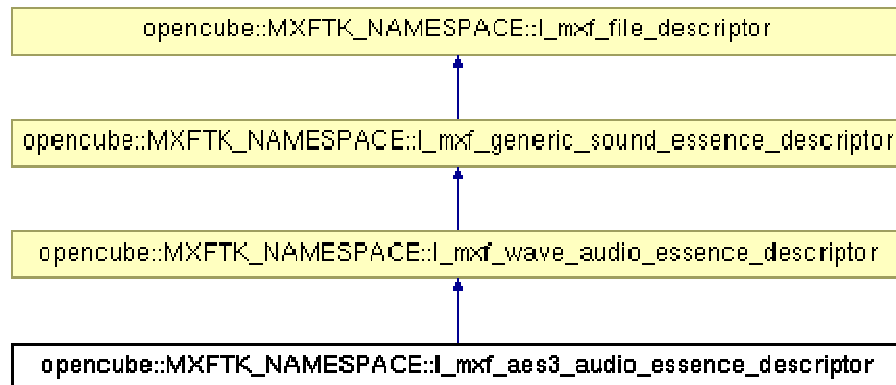
Retrieves a pointer on an **I_mxf_wave_audio_essence_descriptor** interface. Returns *false* if the allocation failed.

bool FreeMxfWaveAudioEssenceDescriptorInterface (I_mxf_wave_audio_essence_descriptor **)

Frees an **I_mxf_wave_audio_essence_descriptor** interface. Returns *true* if successful.

9.9 I_mxf_aes3_audio_essence_descriptor Class Reference

#include <I_mxf_file_descriptor.hpp>



Public Member Functions

void set_emphasis (const uint8_t)

void set_block_start_offset (const uint16_t)

void set_aux_bits_mode (const uint8_t)

void set_channel_status_mode (const uint8_t *, const uint32_t)

void set_fixed_channel_status_data (const uint8_t **, const uint32_t)

void set_user_data_mode (const uint8_t *, const uint32_t)

void set_fixed_user_data (const uint8_t **, const uint32_t)

Constructor and Destructor Documentation

bool NewMxfAes3AudioEssenceDescriptorInterface (I_mxf_aes3_audio_essence_descriptor **)

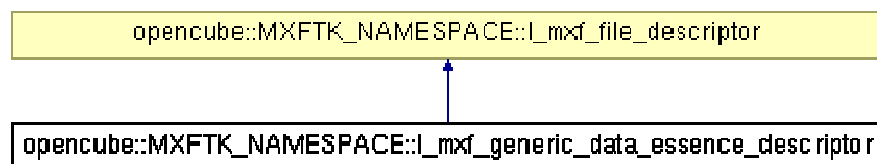
Retrieves a pointer on an **I_mxf_aes3_audio_essence_descriptor** interface. Returns *false* if the allocation failed.

bool FreeMxfAes3AudioEssenceDescriptorInterface (I_mxf_aes3_audio_essence_descriptor **)

Frees an **I_mxf_aes3_audio_essence_descriptor** interface. Returns *true* if successful.

9.10 I_mxf_generic_data_essence_descriptor Class Reference

#include <I_mxf_file_descriptor.hpp>



Public Member Functions

void set_data_essence_coding_ul (const char *)

Constructor and Destructor Documentation

bool NewMxfGenericDataEssenceDescriptorInterface (I_mxf_generic_data_essence_descriptor **)

Retrieves a pointer on an **I_mxf_generic_data_essence_descriptor** interface. Returns *false* if the allocation failed.

bool FreeMxfGenericDataEssenceDescriptorInterface (I_mxf_generic_data_essence_descriptor **)

Frees an **I_mxf_generic_data_essence_descriptor** interface. Returns *true* if successful.

10. Data Handling Interfaces

The following classes help manipulating the data from MXFTk. They are not used to define the structure of an MXF file but rather provide tools useful to their writing and reading.

10.1 concrete_list Class Reference

```
#include <mxftk.hpp>
```

This class describes a list of **I_concrete_material**.

Public Member Functions

void begin ()

I_concrete_material **next ()

int64_t size ()

Member Functions Documentation

void begin ()

Places the internal iterator at the beginning of the list (before the first item of the list).

I_concrete_material **next ()

Increments the internal iterator and returns it. If the returned value is NULL then the end of the list has been reached. The combined use of **begin()** and **next()** enables the traversal of the list. The end is reached as soon as **next()** returns a NULL value.

int64_t size ()

Gets the number of **I_concrete_material** in the list.

10.2 generic_list Class Reference

```
#include <mxftk.hpp>
```

This class describes a list of **I_generic_material**.

Public Member Functions

void begin ()

I_generic_material **next ()

int64_t size ()

Member Functions Documentation

void begin ()

Places the internal iterator at the beginning of the list (before the first item of the list).

const char **next ()

Increments the internal iterator and returns it. If the returned value is NULL then the end of the list has been reached. The combined use of **begin()** and **next()** enables the traversal of the list. The end is reached as soon as **next()** returns a NULL value.

int64_t size ()

Gets the number of **I_generic_material** in the list.

10.3 I_timecode_component Class Reference

```
#include <mxsf_tk.hpp>
```

This class describes the constitutive item of an **I_timecode_material** (timecode track). A timecode track is made of adjacent **I_timecode_component** ordered in a time-linear fashion. Each timecode component defines a starting timecode value and a duration.

Public Member Functions

int64_t get_duration ()

I_timecode *get_timecode ()

Constructor Documentation

bool NewTimecodeComponentInterface (I_timecode_component **, I_timecode *, int64_t)

Retrieves a pointer on an **I_timecode_component** interface to be appended to a timecode material. The **I_timecode** parameter specifies the new timecode as well as its duration measured in edit units of the targeted timecode material. Returns *false* if the allocation failed.

Member Functions Documentation

int64_t get_duration ()

Gets the duration of the current **I_timecode_component** measured in edit units of the **I_timecode_material** it belongs to.

I_timecode *get_timecode ()

Gets the starting timecode value of the current **I_timecode_component**. Returned pointer must not be deleted.

10.4 locators Class Reference

```
#include <mxsf_tk.hpp>
```

This class lets you build a list of source files used upon creation of a new **I_concrete_material**. Note that when you want to wrap in a single track a series of image you should call the function **add()** only once with the appropriate syntax (please refer to **I_concrete_material** documentation)

Public Member Functions

void add (const char **)

void begin ()

I_metadata **next ()

int64_t size ()

Constructor and Destructor Documentation

bool GetEmptyLocatorInterface(locators)**

Retrieves an empty list of locators. Files can be added to this list using the function **add()**. Return *false* if the

allocation failed.

bool FreeLocatorInterface(locators)**

Frees a **locators** list. Return *false* if the deallocation failed.

Member Function Documentation

virtual void add (const char **)

Adds a source file path to the current list.

void begin ()

Places the internal iterator at the beginning of the list (before the first item of the list).

const char **next ()

Increments the internal iterator and returns it. If the returned value is NULL then the end of the list has been reached. The combined use of **begin()** and **next()** enables the traversal of the list. The end is reached as soon as **next()** returns a NULL value.

int64_t size ()

Gets the number of locators in the list.

10.5 crypted_locator_element Class Reference

```
#include <mx_f_tk.hpp>
```

This class lets you specify the location of the source file to be wrapped and crypted in an MXF file. It also provides interfaces to set encryption keys. Note that when you want to wrap in a single track a series of image you should call the function **add()** only once with the appropriate syntax (please refer to **I_concrete_material** documentation)

Public Member Functions

const char *get_location () const

uint64_t get_plaintext_offset () const

uint8_t *get_cryptographic_key_id () const

uint8_t *get_cipher_key () const

Constructor and Destructor Documentation

bool NewCryptedLocatorElementInterface (crypted_locator_element **, const char *, const uint64_t, const uint8_t *, const uint8_t *)

Retrieves an interface on a crypted locator element. The first parameter is the complete path toward the file to be wrapped. The second parameter is the plaintext offset. Finally the two last parameters are respectively the public cryptographic key and the private cipher key to be used for the encryption of this source file.

bool FreeCryptedLocatorInterface(crypted_locators)**

Frees a **crypted_locator_element**. Returns *false* if the deallocation failed.

Member Function Documentation

const char *get_location () const

Return the path towards the file to be wrapped.

uint64_t get_plaintext_offset () const

Return the plain text offset. The frames wrapped within an MXF file are not necessarily entirely crypted. The beginning of the frame might remain not crypted and the end crypted. The plaintext offset (measured in bytes) corresponds to the start of the crypted data within a frame.

uint8_t *get_cryptographic_key_id () const

Return the public cryptographic key.

uint8_t *get_cipher_key () const

Return the private cipher key.

10.6 crypted_locators Class Reference

```
#include <mxmf_tk.hpp>
```

This class lets you build a list of source files (that will be crypted in the MXF file) used upon creation of a new **I_concrete_material**.

Public Member Functions

```
void add (const char **)
```

Constructor and Destructor Documentation

bool GetEmptyCryptedLocatorInterface(crypted_locators)**

Retrieves an empty list of crypted locators. Locators of crypted elements can be added to this list using the function **add()**. Return *false* if the allocation failed.

bool FreeCryptedLocatorInterface(crypted_locators)**

Frees a **crypted_locators** list. Returns *false* if the deallocation failed.

Member Function Documentation

virtual void add (crypted_locator_element)**

Adds a crypted locator element to the current list.

10.7 metadata_list Class Reference

```
#include <mxmf_tk.hpp>
```

This class describes a list of **I_metadata**.

Public Member Functions

```
void begin ()
```

```
I_metadata **next ()
```

```
int64_t size ()
```

Member Functions Documentation

void begin ()

Places the internal iterator at the beginning of the list (before the first item of the list).

const char **next ()

Increments the internal iterator and returns it. If the returned value is NULL then the end of the list has been reached. The combined use of **begin()** and **next()** enables the traversal of the list. The end is reached as soon as **next()** returns a NULL value.

int64_t size ()

Gets the number of **I_metadata** in the list.

10.8 ordered_timecode_list Class Reference

```
#include <mx_f_tk.hpp>
```

This class handles an ordered list of **I_timecode_component**. It is usually retrieved from the method **get_timecode_list()** from **I_timecode_material** and describes a timecode track.

Public Member Functions

void begin ()**I_timecode_component **next ()****int64_t size ()**

Member Functions Documentation

void begin ()

Places the internal iterator at the beginning of the list (before the first item of the list).

const char **next ()

Increments the internal iterator and returns it. If the returned value is NULL then the end of the list has been reached. The combined use of **begin()** and **next()** enables the traversal of the list. The end is reached as soon as **next()** returns a NULL value.

int64_t size ()

Gets the number of **I_timecode_component** in the list.

10.9 ordered_track_item_list Class Reference

```
#include <mx_f_tk.hpp>
```

This class handles an ordered list of **I_track_item**. It is usually retrieved from the method **elements()** from **I_track** and describes the content of a track. Each **I_track_item** of the list can be dynamically cast into one of its possible derivations: **I_source_clip**, **I_dm_source_clip** or **I_dm_segment**.

Public Member Functions

void begin ()**I_track_item **next ()****int64_t size ()**

Member Functions Documentation

void begin ()

Places the internal iterator at the beginning of the list (before the first item of the list).

const char **next ()

Increments the internal iterator and returns it. If the returned value is NULL then the end of the list has been reached. The combined use of **begin()** and **next()** enables the traversal of the list. The end is reached as soon as **next()** returns a NULL value.

int64_t size ()

Gets the number of **I_track_item** in the list.

10.10 properties_list Class Reference

```
#include <mx_fTk.hpp>
```

This class describes a list of **I_property** usually defining the content of an **I_metadata**.

Public Member Functions

void begin ()

I_property **next ()

int64_t size ()

Member Functions Documentation

void begin ()

Places the internal iterator at the beginning of the list (before the first item of the list).

const char **next ()

Increments the internal iterator and returns it. If the returned value is NULL then the end of the list has been reached. The combined use of **begin()** and **next()** enables the traversal of the list. The end is reached as soon as **next()** returns a NULL value.

int64_t size ()

Gets the number of **I_property** in the list.

10.11 rational Class Reference

```
#include <mx_fTk.hpp>
```

This class defines a rational number to specify edit rates.

Public Member Functions

uint32_t get_den ()

double get_double ()

uint32_t get_num ()

Constructor and Destructor Documentation

bool NewRationalInterface(rational, uint32_t, uint32_t)**

Retrieves a pointer on a **rational** interface. The first integer sets the numerator while the second sets the denominator. Returns *false* if the allocation failed.

bool FreeRationalInterface(rational)**

Frees a **rational** interface. Returns *false* if the deallocation failed.

Member Functions Documentation

uint32_t get_den ()

Gets the denominator.

double get_double ()

Gets the closest double value from this rational.

uint32_t get_num ()

Gets the numerator.

10.12 track_list Class Reference

```
#include <mxftk.hpp>
```

This class handles a list of **I_track** usually returned by the functions **source_tracks()** or **metadata_tracks()** from **I_generic_material**. The list returned by **source_tracks()** within an **I_concrete_material** is actually a list of **I_concrete_track**. In this case, the **I_track** from the **track_list** should be statically cast into **I_concrete_track**. Contrary to other list interfaces, this one lets you add and suppress items in order to build up the second parameter of **add_metadata()** from **I_track**.

Public Member Functions

void add (I_track **)

void begin ()

I_track **next ()

void remove (I_track **)

int64_t size ()

Constructor and Destructor Documentation

bool GetEmptyTrackListInterface(track_list)**

Retrieves a pointer on an empty **track_list**. Returns *false* if the allocation failed.

bool FreeTrackListInterface(track_list)**

Frees a **track_list**. It will destroy the list not its items. Returns *false* if the deallocation failed.

Member Functions Documentation

void add (I_track **)

Adds a new track to the current list. The internal iterator is reset to the beginning of the list.

void begin ()

Places the internal iterator at the beginning of the list (before the first item of the list).

const char **next ()

Increments the internal iterator and returns it. If the returned value is NULL then the end of the list has been reached. The combined use of **begin()** and **next()** enables the traversal of the list. The end is reached as soon as **next()** returns a NULL value.

void remove (l_track **)

Removes a track from the current list. The internal iterator is reset to the beginning of the list.

int64_t size ()

Gets the number of **I_track** in the list.

10.13 mxf_file_list Class Reference

```
#include <mxfile_list.hpp>
```

This class handles a list of **I_mxf_file** used with the function **synchronize()** from the **I_opatom_assembler**.

Public Member Functions

void add (I_mxf_file **)**void begin ()****I_mxf_file **next ()****void remove (I_mxf_file **)****int64_t size ()**

Constructor and Destructor Documentation

bool GetEmptyMxfFileListInterface (mxfile_list **)

Retrieves a pointer on an empty **mxfile_list**. Returns *false* if the allocation failed.

bool FreeMxfFileListInterface (mxfile_list **)

Frees an **mxfile_list**. It will destroy the list, not its items. Returns *false* if the deallocation failed.

Member Functions Documentation

void add (I_mxf_file **)

Adds a new file to the current list. The internal iterator is reset to the beginning of the list.

void begin ()

Places the internal iterator at the beginning of the list (before the first item of the list).

const char **next ()

Increments the internal iterator and returns it. If the returned value is NULL then the end of the list has been reached. The combined use of **begin()** and **next()** enables the traversal of the list. The end is reached as soon as **next()** returns a NULL value.

void remove (I_mxf_file **)

Removes a file from the current list. The internal iterator is reset to the beginning of the list.

int64_t size ()

Gets the number of **I_mxf_file** in the list.

11. DCP Creator

DCPCreator is a separate library used to create the XML files which - associated with the right MXF files - form a Digital Cinema Package (DCP).

11.1 Composition Playlist

The following classes are defined in the namespace `opencube::DCP_CREATOR_NAMESPACE::CPL`

11.1.1 I_Marker Class Reference

```
#include <I_compositionplaylist.hpp>
```

This class represents a marker as defined in the chapter 8.3.1 of the SMTE429-7 standard

Public Member Functions

```
bool setLabel(const char *, const char * = 0)
bool setAnnotationText(const char *, const char * = 0)
bool setOffset(int64_t)
```

Constructor and Destructor Documentation

```
bool NewMarkerInterface(I_Marker **)
    Retrieves a pointer on an I_Marker interface. Returns false if allocation failed.

bool FreeMarkerInterface(I_Marker **)
    Frees an I_Marker interface. Returns true if successful.
```

Member Functions Documentation

```
bool setLabel(const char *, const char * = 0)
    Defines the label of the marker. Refer to the table 4 for further information.

bool setAnnotationText(const char *, const char * = 0)
    Free-form, human-readable description (optional).

bool setOffset(int64_t)
    Absolute position of the marker from the start of the marker asset.
```

11.1.2 I_MarkerAsset Class Reference

```
#include <I_compositionplaylist.hpp>
```

This class represents a marker asset as defined in the chapter 8.3.2 of the SMTE429-7 standard

Public Member Functions

```
bool setId(const char *)
```



```
bool setAnnotationText(const char *, const char * = 0)
bool setEditRate(int32_t, int32_t)
bool setIntrinsicDuration(int64_t)
bool setEntryPoint(int64_t)
bool setDuration(int64_t)
```

Constructor and Destructor Documentation

```
bool NewMarkerAssetInterface(I_MarkerAsset **)
```

Retrieves a pointer on an **I_MarkerAsset** interface. Returns *false* if allocation failed.

```
bool FreeMarkerAssetInterface(I_MarkerAsset **)
```

Frees an **I_MarkerAsset** interface. Returns *true* if successful.

Member Functions Documentation

```
bool setId(const char *)
```

UUID of the marker

```
bool setAnnotationText(const char *, const char * = 0)
```

Free-form, human-readable description(optional).

```
bool setEditRate(int32_t, int32_t)
```

Defines the edit rate of the asset.

```
bool setIntrinsicDuration(int64_t)
```

Defines the native duration of the asset.

```
bool setEntryPoint(int64_t)
```

Identifies the edit unit where the playback should start.

```
bool setDuration(int64_t)
```

Defines the duration of the playable region of the asset.

11.1.3 I_PictureTrackFileAsset class reference

```
#include <I_compositionplaylist.hpp>
```

This class represents a picture track file asset as defined in the chapter 8.4 of the SMTE429-7 standard

Public Member Functions

```
bool setFilename(const char *)
bool getFilename(char **, uint32_t *) const
bool freeFilename(char **) const
bool setId(const char *)
bool setAnnotationText(const char *, const char * = 0)
bool setEditRate(int32_t, int32_t)
bool setIntrinsicDuration(int64_t)
bool setEntryPoint(int64_t)
bool setDuration(int64_t)
bool setKeyld(const char *)
```


bool setHash(const char *)
bool setFrameRate(int32_t, int32_t)
bool setScreenAspectRatio(int32_t, int32_t)

Constructor and Destructor Documentation

bool NewPictureTrackFileAssetInterface(I_PictureTrackFileAsset **)
Retrieves a pointer on an **I_PictureTrackFileAsset** interface. Returns *false* if allocation failed.

bool FreePictureTrackFileAssetInterface(I_PictureTrackFileAsset **)
Frees an **I_PictureTrackFileAsset** interface. Returns *true* if successful.

Member Functions Documentation

bool setFilename(const char *)
Sets the Asset filename.

bool getFilename(char **, uint32_t *) const
Gets the Asset filename and its length. Must be deleted by the API user.

bool freeFilename(char **) const
Frees the Asset filename.

bool setUuid(const char *)
UUID of the asset (16 last bytes of the UMID of the source package).

bool setAnnotationText(const char *, const char * = 0)
Free-form, human-readable description (optional).

bool setEditRate(int32_t, int32_t)
Defines the edit rate of the asset.

bool setIntrinsicDuration(int64_t)
Defines the native duration of the asset.

bool setEntryPoint(int64_t)
Identifies the edit unit where playback should start (optional).

bool setDuration(int64_t)
Defines the duration of the playable region of the asset (duration).

bool setKeyId(const char *)
Cryptographic key used to encrypt the track file (optional).

bool setHash(const char *)
Contains the message digest of the track file computed using the SHA-1 algorithm (optional).

bool setFrameRate(int32_t, int32_t)
Frame rate of the track file (optional).

bool setScreenAspectRatio(int32_t, int32_t)
Aspect ratio of the track file (optional).

11.1.4 I_SoundTrackFileAsset class reference

```
#include <I_compositionplaylist.hpp>
```

This class represents a sound track file asset as defined in the chapter 8.5 of the SMTE429-7 standard

Public Member Functions

```
bool setFilename(const char *)
bool getFilename(char **, uint32_t *) const
bool freeFilename(char **) const
bool setId(const char *)
bool setAnnotationText(const char *, const char * = 0)
bool setEditRate(int32_t, int32_t)
bool setIntrinsicDuration(int64_t)
bool setEntryPoint(int64_t)
bool setDuration(int64_t)
bool setKeyId(const char *)
bool setHash(const char *)
bool setLanguage(const char *)
```

Constructor and Destructor Documentation

```
bool NewSoundTrackFileAssetInterface(I_SoundTrackFileAsset **)
```

Retrieves a pointer on an **I_SoundTrackFileAsset** interface. Returns *false* if allocation failed.

```
bool NewSoundTrackFileAssetInterface(I_SoundTrackFileAsset **)
```

Frees an **I_SoundTrackFileAsset** interface. Returns *true* if successful.

Member Functions Documentation

```
bool setFilename(const char *)
```

Sets the Asset filename.

```
bool getFilename(char **, uint32_t *) const
```

Gets the Asset filename and its length. Must be deleted by the API user.

```
bool freeFilename(char **) const
```

Frees the Asset filename.

```
bool setId(const char *)
```

UUID of the asset (16 last bytes of the UMID of the source package).

```
bool setAnnotationText(const char *, const char * = 0)
```

Free-form, human-readable description (optional).

```
bool setEditRate(int32_t, int32_t)
```

Defines the edit rate of the asset.

```
bool setIntrinsicDuration(int64_t)
```

Defines the native duration of the asset.

```
bool setEntryPoint(int64_t)
```

Identifies the edit unit where playback should start (optional).

bool setDuration(int64_t)

Defines the duration of the playable region of the asset (duration).

bool setKeyld(const char *)

Cryptographic key used to encrypt the track file (optional).

bool setHash(const char *)

Contains the message digest of the track file computed using the SHA-1 algorithm (optional).

bool setLanguage(const char *)

Reflects the primary spoken language.

11.1.5 I_SubtitleTrackFileAsset class reference

```
#include <I_compositionplaylist.hpp>
```

This class represents a subtitle track file asset as defined in the chapter 8.6 of the SMTE429-7 standard

Public Member Functions

bool setFilename(const char *)**bool getFilename(char **, uint32_t *) const****bool freeFilename(char **) const****bool setId(const char *)****bool setAnnotationText(const char *, const char * = 0)****bool setEditRate(int32_t, int32_t)****bool setIntrinsicDuration(int64_t)****bool setEntryPoint(int64_t)****bool setDuration(int64_t)****bool setKeyld(const char *)****bool setHash(const char *)****bool setLanguage(const char *)**

Constructor and Destructor Documentation

bool NewSubtitleTrackFileAssetInterface(I_SubtitleTrackFileAsset **)

Retrieves a pointer on an **I_SubtitleTrackFileAsset** interface. Return *false* if allocation failed.

bool FreeSubtitleTrackFileAssetInterface(I_SubtitleTrackFileAsset **)

Frees an **I_SubtitleTrackFileAsset** interface. Return *true* if successful.

Member Functions Documentation

bool setFilename(const char *)

Sets the Asset filename.

bool getFilename(char **, uint32_t *) const

Gets the Asset filename and its length. Must be deleted by the API user.

bool freeFilename(char **) const

Frees the Asset filename.

bool setId(const char *)
UUID of the asset (16 last bytes of the UMID of the source package).

bool setAnnotationText(const char *, const char * = 0)
Free-form, human-readable description (optional).

bool setEditRate(int32_t, int32_t)
Defines the edit rate of the asset.

bool setIntrinsicDuration(int64_t)
Defines the native duration of the asset.

bool setEntryPoint(int64_t)
Identifies the edit unit where playback should start (optional).

bool setDuration(int64_t)
Defines the duration of the playable region of the asset (duration).

bool setKeyId(const char *)
Cryptographic key used to encrypt the track file (optional).

bool setHash(const char *)
Contains the message digest of the track file computed using the SHA-1 algorithm (optional).

bool setLanguage(const char *)
Reflects the primary text language used by the subtitle essence.

11.1.6 I_Reel Class Reference

```
#include <I_compositionplaylist.hpp>
```

This class represents a reel as defined in the chapter 7 of the SMTE429-7 standard

Public Member Functions

bool setId(const char *)
bool setAnnotationText(const char *, const char * = 0)
void setMainMarkers(I_MarkerAsset *)
void setMainPicture(I_PictureTrackFileAsset *)
void setMainSound(I_SoundTrackFileAsset *)
void setMainSubtitle(I_SubtitleTrackFileAsset *)

Constructor and Destructor Documentation

bool NewReelInterface(I_Reel **)
Retrieves a pointer on an **I_Reel** interface. Return *false* if allocation failed.

bool FreeReelInterface(I_Reel **)
Frees an **I_Reel** interface. Return *true* if successful.

Member Functions Documentation

bool setId(const char *)

UUID of the composition reel

bool setAnnotationText(const char *, const char * = 0)

Free-form, human-readable description of the reel (optional).

void setMainMarkers(I_MarkerAsset *)

Defines the markers associated to the theatrical presentation, i.e. MainPicture and MainSound assets (optional).

void setMainPicture(I_PictureTrackFileAsset *)

Defines the picture essence to be projected onto the main screen (optional).

void setMainSound(I_SoundTrackFileAsset *)

Defines the sound essence to be reproduced in the auditorium (optional).

void setMainSubtitle(I_SubtitleTrackFileAsset *)

Defines the subtitle essence to be reproduced on the main screen in the auditorium (optional).

11.1.7 I_CompositionPlaylist Class Reference

#include <I_compositionplaylist.hpp>

This class represents a composition playlist as defined in the chapter 6 of the SMTE429-7 standard

Public Member Functions

bool setFilename(const char *)

bool getFilename(char **, uint32_t *) const

bool freeFilename(char **) const

bool setId(const char *)

bool setAnnotationText(const char *, const char * = 0)

bool setIconId(const char *)

bool setIssueDate(const char *)

bool setIssuer(const char *, const char * = 0)

bool setCreator(const char *, const char * = 0)

bool setContentTitleText(const char *, const char * = 0)

bool setContentKind(const char *, const char * = 0)

bool setContentVersion(const char *, const char *, const char * = 0)

bool addRating(const char * agency, const char * label)

bool addReel(I_Reel *)

void read(const char *)

void write(const char *)

Constructor and Destructor Documentation

bool NewCompositionPlaylistInterface(I_CompositionPlaylist **)

Retrieves a pointer on an **I_CompositionPlaylist** interface. Returns *false* if allocation failed.

bool FreeCompositionPlaylistInterface(I_CompositionPlaylist **)

Frees an **I_CompositionPlaylist** interface. Returns *true* if successful.

Member Functions Documentation

bool setFilename(const char *)

Sets the composition playlist filename.

bool getFilename(char **, uint32_t *) const

Gets the composition playlist filename and its length. Must be deleted by the API user.

bool freeFilename(char **) const

Frees the composition playlist filename.

bool setId(const char *)

UUID of the composition playlist.

bool setAnnotationText(const char *, const char * = 0)

Free-form, human-readable description (optional).

bool setIconId(const char *)

Identifies an external image resource containing a picture icon illustrating the composition playlist (optional).

bool setIssueDate(const char *)

Date and Time at which the composition playlist was issued.

bool setIssuer(const char *, const char * = 0)

Free-form, human-readable description of the person or company who has created the composition playlist (optional).

bool setCreator(const char *, const char * = 0)

Free-form, human-readable description of the system that was used to create composition playlist (optional).

bool setContentTitleText(const char *, const char * = 0)

Human-readable title for the composition playlist.

bool setContentKind(const char *, const char * = 0)

Defines the kind of material referred by the composition playlist.

bool setContentVersion(const char *, const char *, const char * = 0)

Defines the version of material referred by the composition playlist.

bool addRating(const char * agency, const char * label)

Agency represents the agency issuing the rating. Label represents the rating itself.

bool addReel(I_Reel *)

See I_Reel above. Adds a new Reel to the composition playlist.

void read(const char *)

Builds an I_CompositionPlayList instance from an XML file.

void write(const char *)

Writes the instance in an XML file.

11.2 Packing List

The following classes are defined in the namespace `opencube::DCP_CREATOR_NAMESPACE::PKL`

11.2.1 I_Asset Class Reference

```
#include <I_packinglist.hpp>
```

This class represents an asset as defined in the chapter 5 of the SMTE429-8 standard

Public Member Functions

bool setId(const char * value)

bool setAnnotationText(const char * value, const char * attr = 0)

bool setHash(const char * value)

bool setSize(uint64_t value)

bool setType(const char * value)

bool setOriginalFileName(const char * value, const char * attr = 0)

Constructor and Destructor Documentation

bool NewPKLAssetInterface(I_Asset **)

Retrieves a pointer on an **I_Asset** interface. Returns *false* if allocation failed.

bool FreePKLAssetInterface(I_Asset **)

Frees an **I_Asset** interface. Returns *true* if successful.

Member Functions Documentation

bool setId(const char * value)

UUID of the asset.

bool setAnnotationText(const char * value, const char * attr = 0)

Free-form, human-readable description of the asset (optional).

bool setHash(const char * value)

Base64 representation of the SHA-1 message digest of the asset.

bool setSize(uint64_t value)

Length in bytes of the asset.

bool setType(const char * value)

MIME type of the asset

bool setOriginalFileName(const char * value, const char * attr = 0)

File name of the asset at the moment when asset was created.

11.2.2 I_PackingList Class Reference

```
#include <I_packinglist.hpp>
```

This class represents a packing list as defined in the chapter 4 of the SMTE429-8 standard

Public Member Functions

bool setId(const char * value)

bool setAnnotationText(const char * value, const char * attr = 0)


```
bool setIconId(const char * value)
bool setIssueDate(const char * value)
bool setIssuer(const char * value, const char * attr = 0)
bool setCreator(const char * value, const char * attr = 0)
bool setGroupId(const char * value)
bool addAsset(I_Asset *)
void read(const char *)
void write(const char *)
```

Constructor and Destructor Documentation

bool NewPackingListInterface(I_PackingList **)

Retrieves a pointer on an **I_PackingList** interface. Returns *false* if allocation failed.

bool FreePackingListInterface(I_PackingList **)

Frees an **I_PackingList** interface. Returns *true* if successful.

Member Functions Documentation

bool setId(const char * value)

UUID of the packing list.

bool setAnnotationText(const char * value, const char * attr = 0)

Free-form, human-readable description of the packing list (optional).

bool setIconId(const char * value)

Identifies an external image resource containing a picture icon illustrating the packing list (optional).

bool setIssueDate(const char * value)

Date and Time at which the packing list was issued.

bool setIssuer(const char * value, const char * attr = 0)

Free-form, human-readable description of person or company who has created the packing list.

bool setCreator(const char * value, const char * attr = 0)

Free-form, human-readable description of the system that was used to create packing list.

bool setGroupId(const char * value)

Used to group together different packing list (optional).

bool addAsset(I_Asset *)

See **I_Asset** above. Adds an **I_Asset** to the packing list.

void read(const char *)

Builds an **I_PackingList** instance from an XML file.

void write(const char *)

Writes the instance in an XML file.

11.3 Asset Map and Volume Index

The following classes are defined in the namespace `opencube::DCP_CREATOR_NAMESPACE::AM`

11.3.1 I_Chunk Class Reference

```
#include <I_assetmap.hpp>
```

This class represents a chunk as defined in the chapter 6 of the SMTE429-9 standard

Public Member Functions

```
bool setPath(const char * value)
bool setVolumeIndex(uint32_t value)
bool setOffset(uint32_t value)
bool setLength(uint32_t value)
```

Constructor and Destructor Documentation

```
bool NewChunkInterface(I_Chunk **)
```

Retrieves a pointer on an **I_Chunk** interface. Returns *false* if allocation failed.

```
bool FreeChunkInterface(I_Chunk **)
```

Frees an **I_Chunk** interface. Returns *true* if successful.

Member Functions Documentation

```
bool setPath(const char * value)
```

Indicates the complete path for the chunk.

```
bool setVolumeIndex(uint32_t value)
```

Indicates the index of the volume containing the chunk (optional).

```
bool setOffset(uint32_t value)
```

Indicates the offset from the start of the asset to the first byte of the asset segment referenced by this chunk (optional).

```
bool setLength(uint32_t value)
```

Identifies the length in bytes of the chunk (optional).

11.3.2 I_Asset Class Reference

```
#include <I_assetmap.hpp>
```

This class represents an asset as defined in the chapter 5 of the SMTE429-9 standard

Public Member Functions

```
bool setId(const char * value)
bool setAnnotationText(const char * value, const char * attr = 0)
bool setPackingList(int value)
bool addChunk(I_Chunk *)
```

Constructor and Destructor Documentation

bool NewAMAssetInterface(I_Asset **)

Retrieves a pointer on an **I_Asset** interface. Returns *false* if allocation failed.

bool FreeAMAssetInterface(I_Asset **)

Frees an **I_Asset** interface. Returns *true* if successful.

Member Functions Documentation

bool setId(const char * value)

UUID of the asset.

bool setAnnotationText(const char * value, const char * attr = 0)

Free-form, human-readable description of the asset (optional).

bool setPackingList(int value)

Indicates if the asset is a packing list (optional).

bool addChunk(I_Chunk *)

See **I_Chunk** above. Adds an **I_Chunk** to the asset (optional).

11.3.3 I_AssetMap Class Reference

```
#include <I_assetmap.hpp>
```

This class represents an asset map as defined in the chapter 4 of the SMTE429-9 standard

Public Member Functions

bool setId(const char * value)

bool setAnnotationText(const char * value, const char * attr = 0)

bool setCreator(const char * value, const char * attr = 0) = 0;

bool setVolumeCount(uint32_t value)

bool setIssueDate(const char * value)

bool setIssuer(const char * value, const char * attr = 0)

bool addAsset(I_Asset *)

void read(const char *)

void write(const char *)

Constructor and Destructor Documentation

bool NewAssetMapInterface(I_AssetMap **)

Retrieves a pointer on an **I_AssetMap** interface. Returns *false* if allocation failed.

bool FreeAssetMapInterface(I_AssetMap **)

Frees an **I_AssetMap** interface. Returns *true* if successful.

Member Functions Documentation

bool setId(const char * value)

UUID of the AssetMap.

bool setAnnotationText(const char * value, const char * attr = 0)

Free-form, human-readable description of the asset map (optional).

bool setCreator(const char * value, const char * attr = 0) = 0;

Free-form, human-readable description of the system that was used to create asset map.

bool setVolumeCount(uint32_t value)

Total number of volumes referenced by this asset map.

bool setIssueDate(const char * value)

Date and Time at which the asset map was issued.

bool setIssuer(const char * value, const char * attr = 0)

Free-form, human-readable description of person or company who has created the asset map.

bool addAsset(I_Asset *)

See I_Asset above. Adds an I_Asset to the asset map.

void read(const char *)

Builds an I_AssetMap instance from an XML file.

void write(const char *)

Writes the instance in an XML file.

11.3.4 I_VolumeIndex Class Reference

`#include <I_assetmap.hpp>`

This class represents a volume index as defined in the chapter 7 of the SMTE429-9 standard

Constructor and Destructor Documentation

bool NewVolumeIndexInterface(I_VolumeIndex **)

Retrieves a pointer on an I_VolumeIndex interface. Returns *false* if allocation failed.

bool FreeVolumeIndexInterface(I_VolumeIndex **)

Frees an I_VolumeIndex interface. Returns *true* if successful.

Member Functions Documentation

bool setIndex(uint32_t value)

Index of the volume.

void read(const char *) = 0;

Builds an I_VolumeIndex instance from an XML file.

void write(const char *) = 0;

Writes the instance in an XML file.

11.4 DCPCreator

The following class is defined in the namespace `opencube::DCP_CREATOR_NAMESPACE`

11.4.1 Options and creation

void UseXmlNamespace(bool);

Creates XML files with their normative namespace (default is true)

**bool CreateCompositionPlaylist(CPL::I_CompositionPlaylist *,
 const char * filename,
 const char * contentVersionId,
 const char * contentVersionLabel,
 const char * contentTitle = 0,
 I_DCPCreator::CPLContentKind contentKind = I_DCPCreator::CKFeature,
 const char * annotation = 0,
 const char * iconId = 0);**

Refer to I_CompositionPlaylist in I_compositionplaylist.hpp. Builds an instance with all the required fields filled.

bool CreateReel (CPL::I_Reel *, const char * annotation = 0)

Refer to I_Reel in I_compositionplaylist.hpp. Builds an instance with all the required fields filled.

**bool CreateMarker(CPL::I_Marker *,
 I_DCPCreator::CPLStandardMarkerLabel label,
 int64_t offset,
 const char * annotation = 0)**

Refer to I_Marker in I_compositionplaylist.hpp. Builds an instance with all the required fields filled.

**bool CreateMarkerAsset(CPL::I_MarkerAsset *,
 int32_t editRateNum, int32_t editRateDen,
 int64_t intrinsicDuration,
 const char * annotation = 0,
 int64_t entryPoint = 0x7fffffffffffffffULL,
 int64_t duration = 0x7fffffffffffffffULL)**

Refer to I_MarkerAsset in I_compositionplaylist.hpp. Builds an instance with all the required fields filled.

**bool CreatePictureTrackFileAsset (CPL::I_PictureTrackFileAsset *,
 const char * filename,
 const char * instanceId,
 int32_t editRateNum,
 int32_t editRateDen,
 int64_t intrinsicDuration,
 int32_t frameRateNum,
 int32_t frameRateDen,
 int32_t screenARNum,
 int32_t screenARDen,
 const char * annotation = 0,
 int64_t entryPoint = 0x7fffffffffffffffULL,
 int64_t duration = 0x7fffffffffffffffULL,
 const char * keyId = 0,
 const char * hsh = 0)**

Refer to I_PictureTrackFileAsset in I_compositionplaylist.hpp. Builds an instance with all the required fields filled.

**bool CreateSoundTrackFileAsset(CPL::I_SoundTrackFileAsset *,
 const char * filename,
 const char * instanceId,
 int32_t editRateNum,
 int32_t editRateDen,
 int64_t intrinsicDuration,**


```

const char * annotation = 0,
int64_t entryPoint = 0x7fffffffffffffffULL,
int64_t duration = 0x7fffffffffffffffULL,
const char * keyId = 0,
const char * hsh = 0,
const char * lang = 0)

```

Refer to `I_SoundTrackFileAsset` in `I_compositionplaylist.hpp`. Builds an instance with all the required fields filled.

```

bool CreateSubtitleTrackFileAsset(CPL::I_SubtitleTrackFileAsset *,
const char * filename,
const char * instanceId,
int32_t editRateNum,
int32_t editRateDen,
int64_t intrinsicDuration,
const char * annotation = 0,
int64_t entryPoint = 0x7fffffffffffffffULL,
int64_t duration = 0x7fffffffffffffffULL,
const char * keyId = 0,
const char * hsh = 0,
const char * lang = 0)

```

Refer to `I_SubtitleTrackFileAsset` in `I_compositionplaylist.hpp`. Builds an instance with all the required fields filled.

11.4.2 I_DCPCreator class reference

```
#include <I_packinglist.hpp>
```

Simplified interface to build a DCP

Public Member Functions

Constructor and Destructor Documentation

```
bool NewDCPCreatorInterface(I_DCPCreator **)
```

Retrieves a pointer on an `I_DCPCreator` interface. Returns *false* if allocation failed.

```
bool FreeDCPCreatorInterface(I_DCPCreator **)
```

Frees an `I_DCPCreator` interface. Returns *true* if successful.

Member Functions Documentation

```
bool setRootDirectory(const char *)
```

Sets the path where package is built

```
int addCompositionPlayList(CPL::I_CompositionPlayList *)
```

Adds a new composition to the DCP filename of the written CPL XML file required and uncomputable ContentVersion element.

- If not set, required Content Title is set to "Composition Playlist #N" by default.
- If not set, required Content Kind is set to "feature".

```
bool setPackingList(const char *)
```

Sets the relative(-to-root-path) path of the generated Packing List.

- If not set only CPL is generated

void addXmlExtToASSETMAPandVOLINDEX(bool)

AssetMap and Volume Index files should be named ASSETMAP.xml and VOLINDEX.xml but some manufacturers provide DCP files without extension, so you can choose here how to generate them.

void addUUIDToFileNames(bool)

Adds UUID of CPL and PKL to their filename.

bool flush()

Writes DCP.

Related Documentation

enum CPLContentKind

Enumeration members:

CKFeature
CKTrailer
CKTest
CKTeaser
CKRating
CKAdvertisement
CKShort
CKTransitional
CKPsa
CKPolicy

This enumeration defines all Content Kind allowed values.

enum CPLStandardMarkerLabel

Enumeration members:

SMLffoc
SMLlfoc
SMLfftc
SMLlftc
SMLffoi
SMLlfoi
SMLffec
SMLlfec
SMLffob
SMLlfob
SMLffmc
SMLlfmc

This enumeration defines all the allowed Standard Marker Label values.

MXFTk

Classes

```
class concrete_list
class crypted_locator_element
class crypted_locators

class error_list

class generic_list

class l_concrete_material
class l_concrete_track
class l_crypted_essence_stream_task
class l_dcp1_file
class l_dm_segment
class l_dm_source_clip
class l_essence_stream_task
class l_essence_type
class l_evtr_file
class l_generic_material
class l_input_metadata_stream_task
class l_input_mxf_stream_task
class l_input_partial_mxf_stream_task
class l_metadata
class l_metadata_material
class l_mxf_aes3_audio_essence_descriptor
class l_mxf_cdc_i_picture_essence_descriptor
class l_mxf_error
class l_mxf_error_handler
class l_mxf_generic_data_essence_descriptor
class l_mxf_generic_picture_essence_descriptor
class l_mxf_generic_sound_essence_descriptor
class l_mxf_file
class l_mxf_file_descriptor
class l_mxf_jpeg2000_picture_subdescriptor
class l_mxf_mpeg2_video_descriptor
class l_mxf_rgba_picture_essence_descriptor
class l_mxf_subdescriptor
class l_mxf_wave_audio_essence_descriptor
class l_op1a_file
class l_op1b_file
class l_op1c_file
class l_op2a_file
class l_op2b_file
class l_op2c_file
class l_op3a_file
class l_op3b_file
class l_op3c_file
class l_opatom_assembler
class l_opatom_file
class l_output_mxf_stream_task
```



```
class l_property
class l_source_clip
class l_timecode
class l_timecode_component
class l_timecode_material
class l_track
class l_track_item
class l_umid
class l_umid64
class l_value
class l_xdcam_dv_file
class l_xdcam_hd_file
class l_xdcam_imx_file
class l_xdcam_proxy_file
```

```
class locators
```

```
class metadata_list
class mxf_file_list
```

```
class ordered_timecode_list
class ordered_track_item_list
```

```
class properties_list
```

```
class rational
```

```
class track_list
```

Typedefs

```
typedef const char * label_c
```

Functions

```
bool BuildP2XML (const char*)
bool BuildP2XMLBuffer (l_mxf_file*, const char*, size_t*, unsigned char*)
```

```
bool CancelNewP2Shot ()
bool CopyTimecodeInterface (l_timecode **, const l_timecode *)
```

```
void ExtractAesToWave (bool)
void ExtractAes8ToWave (bool)
void ExtractCustomByKLV (bool)
```

```
bool FreeConcreteMaterialInterface (l_concrete_material **)
bool FreeCryptedLocatorInterface (crypted_locators **)
bool FreeCryptedLocatorElementInterface (crypted_locator_element **)
bool FreeDcp1FileInterface (l_dcp1_file **)
bool FreeEvtrFileInterface (l_evtr_file **)
bool FreeLocatorInterface (locators **)
bool FreeMetadataInterface (l_metadata **)
bool FreeMetadataMaterialInterface (l_metadata_material **)
bool FreeMxfAes3AudioEssenceDescriptorInterface (l_mxf_aes3_audio_essence_descriptor **)
```



```

bool FreeMxfCDCIPictureEssenceDescriptorInterface (l_mxf_cdci_picture_essence_descriptor **)
bool FreeMxfFileDescriptorInterface (l_mxf_file_descriptor **)
bool FreeMxfFileInterface (l_mxf_file **)
bool FreeMxfFileListInterface (mxf_file_list **)
bool FreeMxfGenericDataEssenceDescriptorInterface
    (l_mxf_generic_sound_essence_descriptor**)
bool FreeMxfGenericPictureEssenceDescriptorInterface
    (l_mxf_generic_picture_essence_descriptor **)
bool FreeMxfGenericSoundEssenceDescriptorInterface
    (l_mxf_generic_sound_essence_descriptor **)
bool FreeMxfJpeg2000PictureSubdescriptorInterface (l_mxf_jpeg2000_picture_subdescriptor **)
bool FreeMxfMpeg2VideoDescriptorInterface (l_mxf_mpeg2_video_descriptor **)
bool FreeMxfRGBAPictureEssenceDescriptorInterface (l_mxf_rgba_picture_essence_descriptor **)
bool FreeMxfWaveAudioEssenceDescriptorInterface (l_mxf_wave_audio_essence_descriptor **)
bool FreeOp1aFileInterface (l_op1a_file **)
bool FreeOp1bFileInterface (l_op1b_file **)
bool FreeOp1cFileInterface (l_op1c_file **)
bool FreeOp2aFileInterface (l_op2a_file **)
bool FreeOp2bFileInterface (l_op2b_file **)
bool FreeOp2cFileInterface (l_op2c_file **)
bool FreeOp3aFileInterface (l_op3a_file **)
bool FreeOp3bFileInterface (l_op3b_file **)
bool FreeOp3cFileInterface (l_op3c_file **)
bool FreeOpAtomAssemblerInterface (l_opatom_assembler **)
bool FreeOpAtomFileInterface (l_opatom_file **)
bool FreePropertyInterface (l_property **)
bool FreeRationalInterface (rational **)
bool FreeTimecodeInterface (l_timecode **)
bool FreeTimecodeMaterialInterface (l_timecode_material **)
bool FreeTrackListInterface (track_list **)
bool FreeUmidInterface (l_umid **)
bool FreeUmid64Interface (l_umid64 **)
bool FreeValueInterface (l_value **)
bool FreeXdcamDvFileInterface (l_xdcam_dv_file **)
bool FreeXdcamHdFileInterface (l_xdcam_dv_file **)
bool FreeXdcamImxFileInterface (l_xdcam_imx_file **)
bool FreeXdcamProxyFileInterface (l_xdcam_proxy_file **)

bool GetEmptyCryptedLocatorInterface (crypted_locators **)
bool GetEmptyLocatorInterface (locators **)
bool GetEmptyMxfFileListInterface (mxf_file_list **)
bool GetEmptyTrackListInterface (track_list **)
bool GetMxfErrorHandlerInterface (l_mxf_error_handler **)

bool InitMXF_TK (void)
bool InitMXF_TK2 (const char*)

bool NewConcreteMaterialInterface (l_concrete_material **, locators *, wrapping, rational*, bool)
bool NewCryptedLocatorElementInterface (crypted_locator_element **, const char *, const
    uint64_t, const uint8_t *, const uint8_t *)
bool NewCryptedMaterialInterface (l_concrete_material **, crypted_locators *, rational *, bool)
bool NewDcp1FileInterface (l_dcp1_file **, const char *)
bool NewDcp1StreamInterface (l_dcp1_file **, l_output_mxf_stream_task *)
bool NewEvtrFileInterface (l_evtr_file **, const char *)
bool NewEvtrStreamInterface (l_op1a_file **, l_output_mxf_stream_task *)
bool NewExtConcreteMaterialInterface (l_concrete_material **, const char *, rational*, bool)

```



```

bool NewMetadataInterfaceByLabel (I_metadata **, label_c)
bool NewMetadataInterfaceByXML (I_metadata **, const char *)
bool NewMetadataMaterialInterface (I_metadata_material **, metadata_type, I_metadata *, int64_t,
    int64_t, const wchar_t *, size_t)
bool NewMxfAes3AudioEssenceDescriptorInterface (I_mxf_aes3_audio_essence_descriptor **)
bool NewMxfCDClPictureEssenceDescriptorInterface (I_mxf_cdc_picture_essence_descriptor **)
bool NewMxfFileDescriptorInterface (I_mxf_file_descriptor **)
bool NewMxfFileInterface (I_mxf_file **, const char*, mxf_opening_mode)
bool NewMxfGenericDataEssenceDescriptorInterface
    (I_mxf_generic_sound_essence_descriptor**)
bool NewMxfGenericPictureEssenceDescriptorInterface
    (I_mxf_generic_picture_essence_descriptor **)
bool NewMxfGenericSoundEssenceDescriptorInterface
    (I_mxf_generic_sound_essence_descriptor**)
bool NewMxfJpeg2000PictureSubdescriptorInterface (I_mxf_jpeg2000_picture_subdescriptor **)
bool NewMxfMpeg2VideoDescriptorInterface (I_mxf_mpeg2_video_descriptor **)
bool NewMxfRGBAPictureEssenceDescriptorInterface (I_mxf_rgba_picture_essence_descriptor **)
bool NewMxfStreamInterface (I_mxf_file **, I_input_mxf_stream_task *)
bool NewMxfWaveAudioEssenceDescriptorInterface (I_mxf_wave_audio_essence_descriptor **)
bool NewOp1aFileInterface (I_op1a_file **, const char *)
bool NewOp1aStreamInterface (I_op1a_file **, I_output_mxf_stream_task *)
bool NewOp1bFileInterface (I_op1b_file **, const char *)
bool NewOp1bStreamInterface (I_op1b_file **, I_output_mxf_stream_task *)
bool NewOp1cFileInterface (I_op1c_file **, const char *)
bool NewOp1cStreamInterface (I_op1c_file **, I_output_mxf_stream_task *)
bool NewOp2aFileInterface (I_op2a_file **, const char *)
bool NewOp2aStreamInterface (I_op2a_file **, I_output_mxf_stream_task *)
bool NewOp2bFileInterface (I_op2b_file **, const char *)
bool NewOp2bStreamInterface (I_op2b_file **, I_output_mxf_stream_task *)
bool NewOp2cFileInterface (I_op2c_file **, const char *)
bool NewOp2cStreamInterface (I_op2c_file **, I_output_mxf_stream_task *)
bool NewOp3aFileInterface (I_op3a_file **, const char *)
bool NewOp3aStreamInterface (I_op3a_file **, I_output_mxf_stream_task *)
bool NewOp3bFileInterface (I_op3b_file **, const char *)
bool NewOp3bStreamInterface (I_op3b_file **, I_output_mxf_stream_task *)
bool NewOp3cFileInterface (I_op3c_file **, const char *)
bool NewOp3cStreamInterface (I_op3c_file **, I_output_mxf_stream_task *)
bool NewOpAtomAssemblerInterface (I_opatom_assembler **)
bool NewOpAtomFileInterface (I_opatom_file **, const char *)
bool NewOpAtomStreamInterface (I_opatom_file **, I_output_mxf_stream_task *)
bool NewOpZeroFileInterface (I_op1a_file **, const char *)
bool NewOpZeroStreamInterface (I_op1a_file **, I_output_mxf_stream_task *)
bool NewP2Shot (locators*, const char*, const char*, I_timecode*, I_umid*)
bool NewP2ShotFromStream (I_essence_stream_task*, I_essence_stream_task*,
    I_essence_stream_task*, I_essence_stream_task*, I_essence_stream_task*, const char*, const
    char*, I_timecode*, I_umid*)
bool NewPropertyInterface (I_property **, label_c, const I_value *)
bool NewRationalInterface (rational **, uint32_t, uint32_t)
bool NewStreamMaterialInterface (I_concrete_material **, I_essence_stream_task *, wrapping, bool)
bool NewTimecodeComponentInterface (I_timecode_component **, I_timecode *, int64_t)
bool NewTimecodeInterface (I_timecode **, const char *, uint16_t, bool)
bool NewTimecodeInterfaceByValue (I_timecode **, uint16_t, uint16_t, uint16_t, uint16_t, uint16_t,
    bool)
bool NewTimecodeMaterialInterface (I_timecode_material **, int64_t, rational *)
bool NewUmidInterfaceByArray (I_umid **, const uint8_t *)
bool NewUmidInterfaceByString (I_umid **, const char *)

```



```

bool NewUmid64InterfaceByArray (l_umid64 **, const uint8_t *)
bool NewUmid64InterfaceByString (l_umid64 **, const char *)
bool NewValueInterface (l_value **, value_type, size_t, const uint8_t *)
bool NewXdcamDvFileInterface (l_xdcam_dv_file **, const char *)
bool NewXdcamHdFileInterface (l_xdcam_hd_file **, const char *)
bool NewXdcamImxFileInterface (l_xdcam_imx_file **, const char *)
bool NewXdcamProxyFileInterface (l_xdcam_proxy_file **, const char *)
bool NewXdcamDvStreamInterface (l_xdcam_dv_file **, l_output_mxf_stream_task *)
bool NewXdcamHdStreamInterface (l_xdcam_hd_file **, l_output_mxf_stream_task *)
bool NewXdcamImxStreamInterface (l_xdcam_imx_file **, l_output_mxf_stream_task *)
bool NewXdcamProxyStreamInterface (l_xdcam_proxy_file **, l_output_mxf_stream_task *)

```

```
double ProgressNewP2Shot ()
```

```
bool SetDictionary (const char *)
```

```
bool iso7_to_utf16 (wchar_t *, const char *, int)
```

```
l_dm_segment *dm_segment_cast (l_track_item *)
```

```
l_dm_source_clip *dm_source_clip_cast (l_track_item *)
```

```
l_source_clip *source_clip_cast (l_track_item *)
```

```
bool utf16_to_iso7 (char *, const wchar_t *, int)
```

```
bool WaitEndNewP2Shot ()
```

```
void WrapWaveAsAes (bool)
```

```
void WrapWaveAsAes8 (bool)
```

Enumeration Types

anonymous enum

KAG_SIZE

HEADER_REPETITION

INDEX_TABLE

PREFERRED_PARTITION_DURATION

REVERSE_PLAY

enum electro_spatial_form

formulation_two_channel_default

formulation_two_channel

formulation_single_channel

formulation_primary_secondary

formulation_stereophonic

formulation_single_channel_double_frequency

formulation_stereo_left_channel_double_frequency

formulation_stereo_right_channel_double_frequency

formulation_multi_channel_default

formulation_unknown

enum essence_format

for_unknown

for_525_5994p

for_525_5994i
for_525_60i
for_625_50i
for_625_50p
for_720_2398p
for_720_24p
for_720_25p
for_720_2997p
for_720_30p
for_720_50p
for_720_5994p
for_720_60p
for_1080_2398p
for_1080_2398sf
for_1080_24p
for_1080_24sf
for_1080_25p
for_1080_25sf
for_1080_2997p
for_1080_2997sf
for_1080_30p
for_1080_30sf
for_1080_50i
for_1080_50p
for_1080_5994p
for_1080_5994i
for_1080_60i
for_1080_60p

enum essence_source

ess_unknown
ess_d10
ess_d11
ess_dv_unknown
ess_dv_iec
ess_dv_cam_iec
ess_dv_smpte
ess_mpeg_es
ess_mpeg_pes
ess_mpeg_ps
ess_mpeg_ts
ess_mpeg4
ess_uncompressed_unknown
ess_uncompressed_sd
ess_uncompressed_hd1080
ess_uncompressed_hd720
ess_jpeg2k
ess_bwf
ess_aes3
ess_a_law
ess_aiff
ess_vc3

enum gravity

MXF_NONE

MXF_CAUTION
MXF_WARNING
MXF_FATAL

enum opencube::layout

layout_full_frame
layout_separate_fields
layout_single_field
layout_mixed_fields
layout_segmented_frame
layout_unknown

enum metadata_type

m_timeline
m_event
m_static
m_error

enum mxf_opening_mode

Enumeration members:

METADATA_ONLY
CLOSED_METADATA_ONLY
CLOSED_COMPLETE_METADATA
METADATA_AND_LINEAR_PLAYOUT
METADATA_AND_RANDOM_ACCESS

enum sample_structure

samp_unknown
samp_4_1_1
samp_4_2_0
samp_4_2_2
samp_4_4_4_4

enum track_type

picture
sound
data
timeline_metadata
event_metadata
static_metadata
track_type_error

enum value_type

MXF_BOOLEAN
MXF_INT8
MXF_INT16
MXF_INT32
MXF_INT64
MXF_UINT8
MXF_UINT16
MXF_UINT32
MXF_UINT64

MXF_UUID
MXF_UMID32
MXF_UMID64
MXF_RATIONAL
MXF_TIMESTAMP
MXF_ISO7_STRING
MXF_UTF16_STRING
MXF_PRODUCT_VERSION
MXF_STRONG_REF
MXF_WEAK_REF
MXF_ARRAY_BOOLEAN
MXF_ARRAY_INT8
MXF_ARRAY_INT16
MXF_ARRAY_INT32
MXF_ARRAY_INT64
MXF_ARRAY_UINT8
MXF_ARRAY_UINT16
MXF_ARRAY_UINT32
MXF_ARRAY_UINT64
MXF_ARRAY_UUID
MXF_ARRAY_STRONG_REF
MXF_ARRAY_WEAK_REF
MXF_ARRAY_UMID32
MXF_ARRAY_UMID64
MXF_UNKNOWN

enum wrapping

std_wrapping
clip_wrapping
frame_wrapping
line_wrapping
custom_wrapping
evtr_wrapping
xdcam_wrapping
p2_wrapping
k2_wrapping
opzero_wrapping
dcp_wrapping
wrapping_error

Error Messages

// Caution

#1001# <Essence not supported in MxfTk Evaluation version>
#1002# <Cannot proceed: Trying to add/remove metadata to/from a picture, sound or data track>
#1003# <Cannot proceed: Metadata material type does not match metadata track type>
#1004# <Cannot proceed: Going beyond documented tracks scope>
#1005# <Cannot proceed: Going out of timecode boundaries>
#1006# <Cannot proceed: Trying to write data on a closed stream>
#1007# <Cannot proceed: Label not found in dictionary>
#1008# <Cannot proceed: Unexpected type for this label>
#1009# <Cannot proceed: Generic material from each OpAtom file must contain the same number of tracks to be serializable>
#1010# <Cannot proceed: Cannot serialize video tracks. Each OpAtom files must contain maximum one and only one video track>

#1011# <Cannot proceed: Cannot serialize video tracks. Tracks have a different edit rate>
#1012# <Cannot proceed: Cannot serialize audio tracks. Each OpAtom files must contain maximum one and only one audio track>
#1013# <Cannot proceed: Cannot serialize video tracks. Tracks have a different edit rate>
#1014# <Cannot proceed: Cannot serialize data tracks. Each OpAtom files must contain maximum one and only one data track>
#1015# <Cannot proceed: Cannot serialize data tracks. Tracks have a different edit rate>
#1016# <Cannot proceed: Don't know how to serialize event metadata tracks>
#1017# <Cannot proceed: Cannot serialize static metadata tracks. Each OpAtom files must contain maximum one and only one static metadata track>
#1018# <Cannot proceed: Cannot serialize static metadata tracks. Tracks have a different edit rate>
#1019# <Cannot proceed: Cannot serialize timeline metadata tracks. Each OpAtom files must contain maximum one and only one timeline metadata track>
#1020# <Cannot proceed: Cannot serialize timeline metadata tracks. Tracks have a different edit rate>
#1021# <Cannot proceed: Valid call only while streaming in an MXF file>
#1022# <Cannot proceed: A concrete material is already set. Only one is allowed in an OpAtom file>
#1023# <Cannot proceed: There should be one and only essence track in an OpAtom concrete material>
#1024# <Cannot proceed: A concrete material is already set. Only one is allowed in an Op1a file>
#1025# <Cannot proceed: Operational Pattern not supported by this version of MxfTk>
#1026# <Cannot proceed: Only Op1a files are supported in this MxfTk standard version>
#1027# <Cannot proceed: Clip wrapping is not allowed when streaming out an MXF file>
#1028# <Cannot proceed: Invalid duration>
#1029# <Label not found>
#1030# <Cannot proceed: Cannot add this property to the current metadata>
#1031# <Cannot proceed: Timecode track duration doesn't match essence tracks duration>
#1032# <Cannot proceed: Invalid number of essence streams>
#1033# <Cannot proceed: No essence stream to write to>
#1034# <Cannot proceed: Documented tracks are not member of this material>
#1035# <Cannot proceed: At least one of the files is not an OpAtom file>
#1036# <Cannot proceed: Feature supported only in MxfTk advanced version>
#1037# <Cannot proceed: The file must be opened in editing mode to perform this task>
#1038# <Cannot proceed: More than one generic material (material package) was detected.>
#1039# <Cannot proceed: These OpAtom files don't share the same generic material.>
#1040# <Cannot proceed: Output timecode must be continuous. There should be one and only one timecode component.>
#1041# <Cannot proceed: Evtr file should be built using a D10 source.>
#1042# <Cannot proceed: Xdcam proxy file should be built using MPEG4 and A-law sources.>
#1043# <Cannot proceed: Xdcam Dv file should be built using DV IEC and AES3 sources.>
#1044# <Cannot proceed: P2 files should be built using 1 DV IEC or SMPTE source and 2 or 4 AES sources.>
#1045# <Cannot proceed: Basename for P2 files should be exactly 6 characters long + NULL termination.>
#1046# <Cannot proceed: Origin of the metadata track does not fall within the essence track's scope.>
#1047# <Cannot proceed: Metadata segment starts before the origin of the event track.>
#1048# <Cannot proceed: PREFERRED_PARTITION_SIZE can be set only in Op1a file. Use PREFERRED_PARTITION_DURATION instead.>
#1049# <Cannot proceed: You should set the duration of each timecode component when setting a discontinuous timecode.>
#1050# <Cannot proceed: Metadata material cannot be removed from a timeline metadata track.>
#1051# <Cannot proceed: There is not enough digits to represent the first or last number.>
#1052# <Cannot proceed: Invalid first/last/step argument for uncompressed images.>
#1053# <Cannot proceed: External reference file was not set.>
#1054# <Cannot proceed: You cannot set an external reference with an OpZero file.>
#1055# <Cannot proceed: You cannot set an external reference with a DCP file.>
#1056# <Cannot proceed: Edit rate of DCP concrete material must be 24/1 or 48/1.>
#1057# <Cannot proceed: Invalid task for the selected opening mode.

// Warning

#2001# <Invalid File Name, check path>
 #2002# <Error while opening/creating file or directory>
 #2003# <Error while closing file>
 #2004# <No file to work with>
 #2005# <Error while writing file, check disk space>
 #2006# <Seeking beyond file's scope>
 #2007# <Unrecognized MXF Key>
 #2008# <Unrecognized MXF Type>
 #2009# <Unrecognized MXF Descriptor name>
 #2010# <Invalid MPEG2 Frame>
 #2011# <Invalid MXF Wrapping Option>
 #2012# <Invalid Sampling Frequency (MPEG2 Audio ES)>
 #2013# <Cannot Multiplex D10 and AES3-8 streams in an MXF file>
 #2014# <DV encoding not supported>
 #2015# <No data to be processed>
 #2016# <Not supported essence or combination of essences>
 #2017# <Cannot access essences for reading upon creation of an MXF file>
 #2018# <Streaming metadata is not allowed while processing OpAtom files>
 #2019# <No Essence Data to be written>
 #2020# <D10 Essence requires a KAG Size set to 512>
 #2021# <Unknown value_type>
 #2022# <Invalid NULL pointer>
 #2023# <Invalid MXF file structure: Sequence from a track is missing>
 #2024# <Unrecognized track type>
 #2025# <Cannot proceed: Incomplete timecode definition>
 #2026# <Invalid timecode format detected>
 #2027# <Invalid timecode format : illegal drop frame timecode value>
 #2028# <No header metadata in the file: cannot decode MXF>
 #2029# <Invalid Header metadata: preface not found>
 #2030# <Invalid Header metadata: cannot find content storage set>
 #2031# <Invalid Header metadata: unrecognized package>
 #2032# <Invalid Header Metadata: an essence referenced by the the header metadata cannot be found in the file>
 #2033# <Invalid Header Metadata: an essence container is missing a UMID>
 #2034# <Invalid Header Metadata: cannot find essence container>
 #2035# <No essence container data to update indexsid>
 #2036# <Data cannot be linked to header metadata. This MXF file is probably malformed. >
 #2037# <Cannot build metadata tree : unrecognized property>
 #2038# <Broken link: InstanceUID not found>
 #2039# <Cannot build metadata tree : several possible roots detected>
 #2040# <Cannot build metadata tree : no root detected>
 #2041# <Broken link: cannot reach referenced child>
 #2042# <Cannot not build streaming buffer>
 #2043# <Invalid BWF file>
 #2044# <Cannot fill essence klv: custom klv size is too short>
 #2045# <Cannot proceed: Concrete Material should be Xdcam-wrapped.>
 #2046# <Cannot proceed: Clip wrapped sources should have the same duration.>
 #2047# <Cannot proceed: Invalid AES frame size computed.>
 #2048# <Truncated KLV Key>
 #2049# <Truncated KLV Length>
 #2050# <Truncated KLV Value>
 #2051# <Cannot perform a partial restore : header metadata misses duration data.>
 #2052# <Cannot proceed: No data to be retrieved for these timecode values.>
 #2053# <Cannot proceed: This is not a P2 file or it is not stored in a P2 directory structure.>
 #2054# <Cannot proceed: This is not a P2 Video file.>
 #2055# <Cannot proceed: Bit rate of D10 video must be 50, 40 or 30Mbps>
 #2056# <Flush cancelled: You need to provide 1 concrete material to create op1a files.>

#2057# <Flush cancelled: You need to provide 1 concrete material for each opatom file.>
#2058# <Flush cancelled: You need to provide at least 2 concrete materials to create op1b files.>
#2059# <Flush cancelled: You need to provide at least 2 concrete materials to create op2a files.>
#2060# <Flush cancelled: You need to provide at least 4 concrete materials to create op2b files.>
#2061# <Cannot add concrete material: only frame wrapped material can be added to this file.>
#2062# <Cannot link this concrete material with the previous : check that each track is in correspondency with a track of the same nature (video, audio or data).>
#2063# <Cannot proceed : op2b files require at least two set of ganged concrete material.>
#2064# <Cannot proceed : each set of ganged material should contain the same number of concrete material in an op2b file.>
#2065# <Cannot proceed : you must create a new set of ganged material first.>
#2066# <Cannot proceed : a thread is already running for that file.>
#2067# <Cannot proceed : D10 video system is PAL(NTSC) while D10 audio system is NTSC(PAL).>
#2068# <Cannot proceed : MPEG audio substream should be MPEG ES or AES3 (S302M) only.>
#2069# <Cryptod essence : invalid cipher key.>
#2070# <Encryption failed.>
#2071# <Cannot proceed : invalid frame header.>
#2072# <Cannot proceed : uncompressed data, user has to fill file descriptor.>
#2073# <Cannot proceed : file descriptor is not complete.>
#2074# <Cannot proceed : a wrong file descriptor has been set for an essence.>
#2075# <Cannot proceed : user file descriptor doesn't match corresponding essence.>
#2076# <Cannot proceed : unable to reach information in the stream.>
#2077# <Cannot proceed : invalid TIFF header.>
#2078# <Cannot proceed : not supported TIFF format.>
#2079# <Streaming : stream doesn't exist (invalid stream id)>
#2080# <Failed to open externally referenced essence file>
#2081# <Externally referenced essence file is not of the appropriate type>
#2082# <Externally referenced essence file is truncated>
#2083# <Cannot proceed: Failed to build complete path to external reference file.>
#2084# <Cannot proceed: Could not find source package with the same UMID in the external reference file.>
#2085# <Cannot proceed: Invalid generic material for this operational pattern.>
#2086# <Cannot proceed: Invalid timecode for this track.>
#2087# <Cannot proceed: Error during creation of file: not enough space to write index table.>
#2088# <Cannot proceed: Seeking on the output stream must be enabled when creating an OpZero file.>
#2089# <Cannot proceed: D10 audio must be set to 48000Hz.>

// Fatal Error

#3001# <MxfTk Internal Error>
#3002# <Error upon loading of localtag dictionary>
#3003# <MXF_HOME is not set>
#3004# <Xerces Internal Error>
#3005# <Xerces Unexpected Error>
#3006# <No Metadata Dictionary Set>
#3007# <DEFAULT_MXF_DICTIONARY is not set>
#3008# <Error upon loading of MxfTk license file : cannot open file>
#3009# <Error upon loading of MxfTk license file : more than one file detected>
#3010# <Wrong license key in the license file.>
#3011# <Cannot initialize MxfTk library: Check your MXF_HOME path>
#3012# <Cannot complete MxfTk library initialization>
#3013# <MxfTk evaluation version expired>
#3014# <Invalid Metadata dictionary>
#3015# <Invalid MXF file : missing Instance UID properties>
#3016# <MXF does not allow to clip wrap source files larger than 4Gb. Try frame wrapping>
#3017# <MXFTk tried to seek backward on a non-seekable stream.>
#3018# <A numerical error occured. Cannot continue.>
#3019# <Process stopped: The timecode duration does not match the essence tracks duration.>

#3020# <Process stopped: A timeline metadata track does not match the video or audio track duration.>
#3021# <Process stopped: An event metadata track contains a metadata segment that goes beyond the video or audio track duration.>
#3022# <Process stopped: The essence configured by the user does not match the stream data.>
#3023# <Process stopped: The DV audio stream configured by the user is missing from the stream.>
#3024# <Process stopped: The DV audio stream was not properly configured.>
#3025# <No license file to look for.>
#3026# <Cannot find license key in the license file.>
#3027# <License key is truncated in the license file.>
#3028# <Inappropriate license key found in the license file (it is not a key for this product).>
#3029# <Streaming mode : you must define the essence types of the streams in
I_essence_stream_task::get_essence_source().>
#3030# <DV should be set to have no audio when wrapping P2 files.>
#3031# <Invalid KLV : should be a composite one.>
#3032# <Problem while updating track duration.>
#3033# <Unable to create partition.>
#3034# <Invalid license key for this computer. This key was created on another computer or you changed your computer hardware.>

References

377M: SMPTE, MXF File Format,
EG41: SMPTE, MXF Engineering Guideline,
EG42: SMPTE, MXF Descriptive Metadata Engineering Guideline,
380M: SMPTE, MXF Descriptive Metadata Scheme 1,
330M: SMPTE, Unique Material Identifiers,
S429-7: SMPTE, D-Cinema Packaging – Composition Playlist
S429-8: SMPTE, D-Cinema Packaging – Packing List
S429-9: SMPTE, D-Cinema Packaging – Asset Mapping and File Segmentation
RP210: SMPTE, Metadata Dictionary.
