# EBU MXF SDK v2.0:
# EBUCore metadata and Analyzer functionality

This document describes the functionality and structure of the EBU MXF SDK for EBUCore metadata processing, serialization and extraction, and for the analysis of MXF files. It also describes how to use the SDK functions in derivative projects.

## 1 Functionality – EBUCore metadata

The EBU MXF SDK offers functionality for processing EBUCore metadata, and in particular, dealing with the inclusion and extraction of EBUCore metadata in Material Exchange Format (MXF) audiovisual essence container files.

### 1.1 Concepts of EBUCore Processing

This section describes the EBUCore processing functionality of the SDK (illustrated in Section ❶ in Figure 1). The SDK can read and write two representations of EBUCore; the XML variant is read from and written to XML documents that conform to the EBUCore XML schema[1], the MXF variant is read and written to KLV packets, the native encoding of information units in MXF files, that conform to the Class 13 SMPTE metadata element and group dictionary register for EBUCore. For both XML and MXF representations, the EBUCore metadata is read (or written to) an in-memory representation (i.e., an instantiated object model) first and then translated to the other representation through a bi-directional mapping which translates peculiarities between both sides.

The source code that defines the object model for both representations has been generated automatically. It should be again generated when modifications are done to the source definitions, either to the EBUCore XML schema or the MXF-KLV metadata dictionary. Instructions as to how this is done are provided in Appendix 1.

#### 1.1.1 Writing EBUCore metadata into MXF files

Concerning the serialization of KLV EBUCore packets into MXF files, the SDK can function in one of two modes; as part of the wrapping process of raw essence into a new MXF file, and as part of the extension process of an existing MXF file.

Firstly, and illustrated in Section ❷ in Figure 1, the SDK can operate in a mode where a new MXF file is constructed from a number of essence tracks (delivered in separate files) and the EBUCore KLV packets are mixed in with the newly constructed metadata (incl., track structure and essence characteristics) of the MXF file. In this case, the metadata is written as part of the header partition, which is marked closed and complete. There is no need for metadata to be written to any of the body or footer partitions.

The **raw2bmx** example program in ebu-bmx/apps/raw2bmx demonstrates the use of this first mode of operation. Along with the parsing of raw essence and inclusion of its structural metadata in a

[1] The EBU SDK employs the version 1.4 (dated 07/01/2013) of the EBUCore XML schema.

newly written MXF file, the EBUCore metadata is processed from an XML document, hooked onto the MXF file metadata model, and then written to the file along with the other KLV packets in the MXF container.

In the second mode of operation, the SDK writes EBUCore metadata into an existing MXF file, the path depicted in Section ❸ in Figure 1. This mode requires more complex application logic, as the existing file must 1) be modified as efficiently as possible, and 2) the existing metadata must be modified in such a way as to remain fully compliant with the MXF file format specification.
MXF files may carry multiple instances of the file's metadata (each new file partition can contain an updated set of metadata). This way, streaming and growing file scenarios can be supported in which increasingly accurate metadata is continuously inserted as the file being is extended, resulting in an MXF file that contains the most complete metadata in its footer partition. Partitions marked as open and incomplete can instruct MXF interpreters to ignore early sets of metadata and only consider a final closed and complete metadata set as the definitive MXF file structure description.
Unless explicitly instructed otherwise, the SDK uses this mechanism to append the updated metadata in the footer partition of the MXF file. This involves a rewrite of only the footer partition, which requires only limited writing operations since footer partitions contain no essence. Most of the header and (bulky) body partitions remain unchanged, except for an update of the small partition header KLV pack to signal an – as of now – open and incomplete metadata set. Note that, when selecting the metadata to extend, the SDK also interprets partition flags to select only the finalized metadata for extension with EBUCore elements.

Considering the complexity of the MXF file format specification, it is not unlikely that certain implementations of MXF interpreters will lack support for selection of metadata beyond the header partition, and will expect this partition to contain only a single complete metadata set. To support these systems, the SDK can be explicitly instructed to write the EBUCore metadata to the header partition, at the expense of a byte shift operation across the remainder of the MXF file.

The **ebu2mxf** example program in ebu-bmx/apps/ebu2mxf demonstrates the use of this second mode of operation. An existing MXF file is opened, and the EBUCore metadata is appended to its most appropriated (closed and complete metadata where available) set of metadata. This mode uses an identical EBUCore processing path as in the first operation mode, but only attempts to modify only the file metadata without rewriting its essence. The options for this command are as follows:

```
Usage: ebu2mxf <<options>> <filename>
Options:
 -h | --help           Show usage and exit
 -v | --version        Print version info
 -l <file>             Log filename. Default log to stderr/stdout
 -i                    Print file information to stdout
 --ebu-core <file>     Write embedded EBU Core metadata to file
 --force-header        Force metadata to be appended into the header partition
 --dark                Write EBU Core metadata into a dark metadata set
 --sidecar             Write EBU Core metadata as a side-car reference
 --dark-key            Use this custom dark metadata key for metadata embedding.
                       The provided key should a SMPTE UL, formatted as a 'urn:smpte:ul:...'.
 --rp2057              Embed EBU Core metadata according to SMPTE RP 2057 XML embedding.
                       If the XML size is less than 64KB and uses UTF-8 or UTF-16 encoding
                       (declared in the XML prolog) then the XML data is included in the header
                       metadata. Otherwise a Generic Stream partition is used to hold the XML data.
 --remove              Remove EBU Core metadata from the MXF file header metadata
SMPTE RP2057-based XML embedding options:
```

```
--xml-scheme-id <id>  Set the XML payload scheme identifier associated with
                      the RP-2057-embedded metadata.
                      The <id> is one of the following:
                        * a SMPTE UL, formatted as a 'urn:smpte:ul:...',
                        * a UUID, formatted as a 'urn:uuid:...' or as
                          32 hexadecimal characters using a '.' or '-' separator.
                      A default EBU Core scheme identifier is used if this option is not provided.
--xml-lang <tag>      Set the RFC 5646 language tag associated with the RP-2057-embedded metadata.
                      Defaults to the xml:lang attribute in the root element
                      or empty string if not present.
```

To use the SDK in this mode, use one of the `EBUCore::EmbedEBUCoreMetadata()` functions that are documented in the Doxygen documentation.

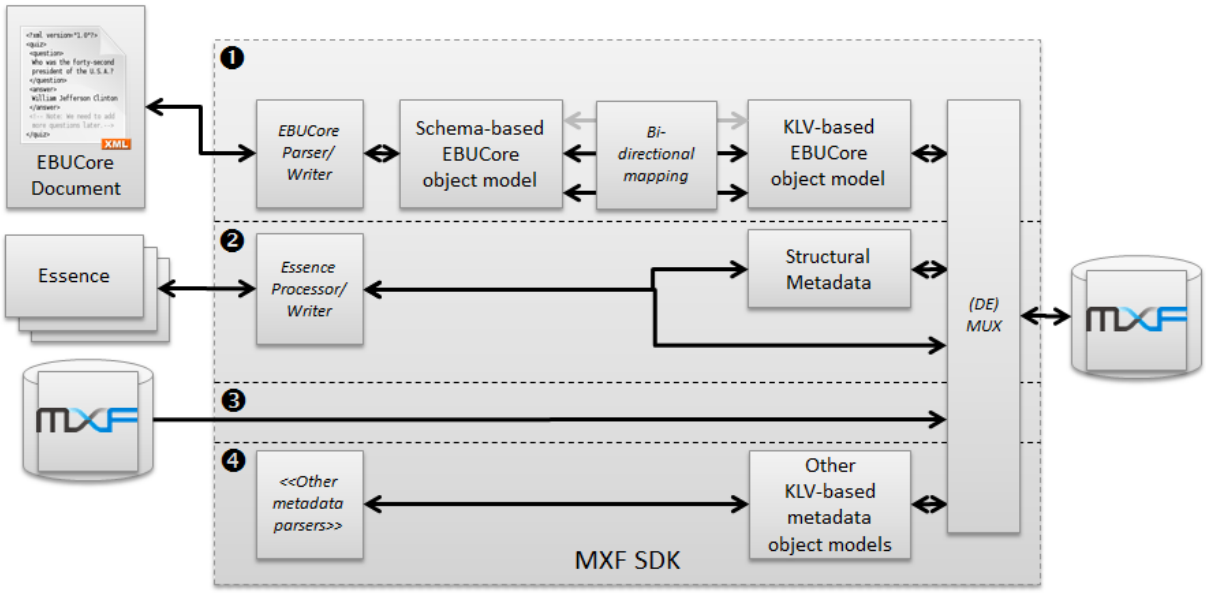


Figure 1: EBU SDK block diagram for EBUCore metadata functionality.

### 1.1.2 Methods of EBUCore serialization

Apart from the overall operation mode of the SDK (writing a new file, or modifying an existing one) and the location where to write the metadata (forcibly in the header partition, or more efficiently in the footer), there are four ways in which the SDK can write actual EBUCore metadata into an MXF file. EBUCore can be embedded as a full object tree of KLV packets, as a minimal number of KLV packets that refer to the external file as side-car metadata, or the XML representation can be inserted into a single 'dark' KLV packet at the end of the header metadata. Finally, a text-based embedding method compliant with SMPTE RP 2057 is now also supported. All methods are discussed in the following paragraphs.

#### *1.1.2.1 KLV Serialization*

EBUCore can be written into the MXF file using a comprehensive tree of KLV packets, formatted according to the MXF specification best practices and the EBUCore dictionary registers; each logical object is contained in a single local set-formatted KLV packet and MXF metadata strong references hold these objects together. Also, each EBUCore KLV object extends the standard MXF *Interchange Object* such that all written metadata objects can, amongst others, participate in the MXF modification tracking mechanism.

Additionally, the EBUCore metadata is inserted into the MXF metadata in such a way that in properly interacts with the timeline model of the MXF file format, as illustrated in Figure 2. Alongside the timeline tracks that describe the essence, a Descriptive Metadata static track is inserted, which contains a reference to the entry point of the EBUCore metadata: an *ebucoreMainFramework* instance. *Part* definitions described in more complex EBUCore documents are also properly modelled on the MXF timeline by using a Descriptive Metadata *Event Track* on which temporal segments with a reference to an *ebucorePartFramework* object instance are assigned for each of the *Part* elements.

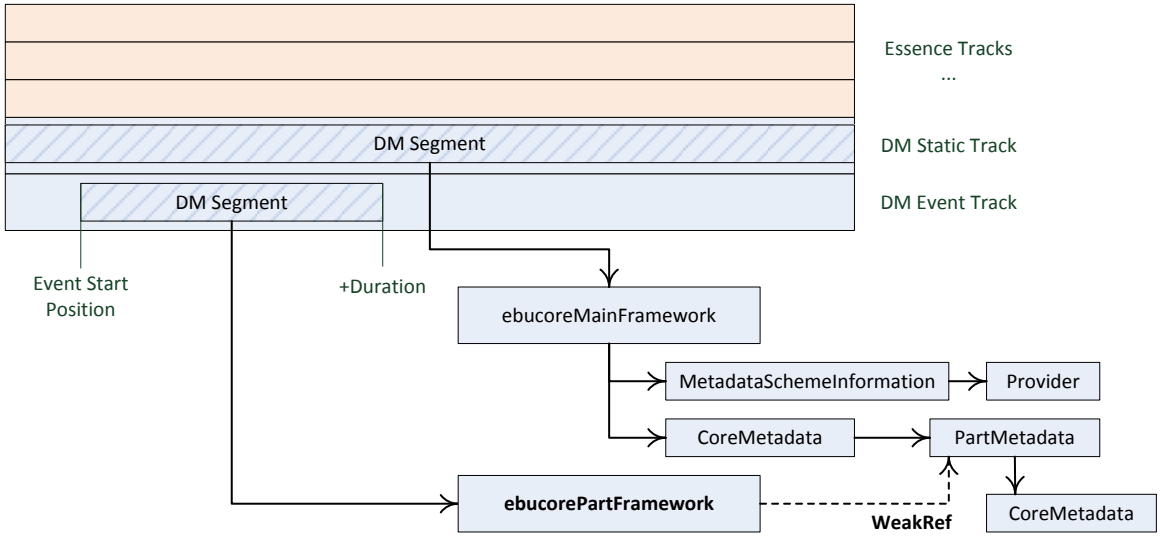


Figure 2: EBUCore metadata and the MXF timeline.

### 1.1.2.2 Side-car serialization

A second method of serialization writes only a minimal set of KLV packets to the MXF file, and uses a reference to an external file in which the actual metadata is stored. This method can be used in scenarios where metadata updates will occur and the recurring modification of MXF files is not feasibly. The downside of this approach is that metadata file must be transferred and kept together with the MXF file throughout the production and distribution process.

With respect to the KLV packets stored in this case, only the timeline elements (the *DM Static Track*, and its segments) and the *ebucoreMainFramework* are written. Also, no *MetadataSchemeInformation* or *CoreMetadata* objects are present. The *ebucoreMainFramework* only contains a *DocumentLocator* field into which the location of the side-car file is written.

### 1.1.2.3 SMPTE RP2057-based serialization

In version 2.0 of the SDK, metadata embedding according to SMPTE RP2057:2011 has been added. This recommended practice defines a generic way of embedding XML-based content into MXF files, without the need for extensive KLV modelling. Similar to side-car serialization, a basic metadata framework is attached to the MXF timeline and this framework points to the entire XML content in a single KLV packet, or if the XML exceeds the size of a KLV local set, the XML content is embedded in a dedicated Generic Stream partition in the MXF file. The SDK supports metadata RP2057-based embedding using a set of default schemes for EBUCore embedding, but can also be used with custom metadata schemes for other metadata embedding applications.

#### *1.1.2.4 'Dark' serialization*

The third method of EBUCore serialization involves embedding a single 'dark' KLV packet in which the EBUCore XML metadata document is written as-is. The KLV packet is inserted as the last packet at the end of the regular header metadata and is identified by a specific EBUCore 'dark' metadata key. No further modifications are done to the MXF file metadata.

In version 2.0 of the SDK, users can specify custom dark metadata keys, in addition to the standard EBUCore keys used when no overrides are given.

### 1.1.3 Extraction of EBUCore metadata from MXF files

The EBU MXF SDK also operates in the reverse direction of the serialization functionality explained above. EBUCore metadata can be extracted from MXF KLV packets and translated to the XML representation, by following the reverse path of Section ❶ in Figure 1. Just as in the serialization direction, the EBUCore metadata is extracted from the most appropriate header metadata (closed and complete where available). The EBUCore metadata is then located by searching through the structural metadata and MXF timeline model, as described in the next section.

When parsing the MXF file, the SDK automatically detects the EBUCore metadata from each of the used serialization methods. It first attempts to read fully KLV-encoded metadata, then tries to locate a side-car metadata file (if applicable), proceeds to locating EBUCore metadata as SMPTE RP2057-compliant metadata, and finally searches for the EBUCore dark metadata packet. The first set of metadata found is employed.

The **mxf2ebu** example program in ebu-bmx/apps/mxf2ebu demonstrates the use of EBUCore metadata extraction. An existing MXF file is opened, and the EBUCore metadata is read from its metadata. The options for this command are as follows:

```
Usage: mxf2ebu.exe <<options>> <filename>
Options:
 -h | --help           Show usage and exit
 -v | --version        Print version info
 -l <file>             Log filename. Default log to stderr/stdout
 -i                    Print file information to stdout
 --ebu-core <file>     Write embedded EBU Core metadata to file
 --dark-key            Use this custom dark metadata key when searching for dark embedded metadata.
                       The provided key should a SMPTE UL, formatted as a 'urn:smpte:ul:...'.
 --xml-scheme-id <id>  Use this XML scheme when searching for relevant RP2057-embedded metadata.
                       The <id> is one of the following:
                        * a SMPTE UL, formatted as a 'urn:smpte:ul:...',
                        * a UUID, formatted as a 'urn:uuid:...' or
                          as 32 hexadecimal characters using a '.' or '-' separator,
                       A default EBU Core scheme identifier is used if this option is not provided.
```

To use the SDK for EBUCore extraction, use one of the `EBUCore::ExtractEBUCoreMetadata()` functions that are documented in the Doxygen documentation.

# 2 Functionality – Analyzer

The EBU MXF SDK offers functionality for analyzing MXF files, including its metadata and other file structures such as index tables, and reporting about this analysis in the standardized SMPTE ST-434 XML format.

The analysis methodology closely follows the types of reports defined in the ST-434 standard. There are two formats to distinguish:

- Metadata only. This report format contains a representation of a single set of header metadata and contains an XML derivation of all metadata sets in the header metadata, including generic elements for dark properties (in known sets) or dark sets (of which the definition is dark to the analyzer in its entirety).
  Within this metadata-only format, two methods are defined to structure related metadata sets. One method ('logical') writes metadata sets in a tree-like fashion, in which strong references are resolved and replaced in-line by the referenced object, creating a structure that represents the logical relationships between metadata elements. The other method ('physical') lists all metadata sets one after the other at the order they physically appear in the MXF file and writes strong references as unresolved references using the target InstanceUID field of the destination metadata set.
- Full physical multiplex information. This report format contains a representation of the entire MXF file container (or multiplex), in addition to a report concerning the header metadata. This format also lists low-level metadata support packs (e.g., primer pack), partition information (e.g., partition packs and the Random Index Pack), and index tables.

When invoking the SDK Analysis functions each variant of these formats can be selected as the reported output (i.e., METADATA or MUX, and LOGICAL or PHYSICAL).

For version 2.0 of the SDK, a new strict analysis option was added. When active, it will restrict the output reports to only elements explicitly defined in the ST-434 standard. When left out, the analysis code will also output information about elements not strictly defined by the standard, but known by the SDK (this can be useful in cases that the SDK libraries have been updated with new initiatives that have not yet been included in a revision of ST-434).

The **mxfanalyzer** example program in ebu-mxfsdk/Analyzer/apps/analyzer demonstrates the use of the SDK's analyzer functions. An existing MXF file is opened, analyzed, and the report is written in the format selected by a number of command line options, as follows:

```
Usage: mxfanalyzer.exe <<options>> <filename>
Options:
 -h | --help           Show usage and exit
 -v | --version        Print version info
 --report <file>       Write analysis report to file
 --physical            Output a physical layout of the header metadata
 --logical             Output a logical layout of the header metadata (default)
 --metadata            Perform only a metadata analysis (default)
 --mux                 Perform an analysis on the entire MXF file mux
 --deepindex           Perform a deep index table analysis
 --strict              Perform a strict, fully ST-434-compliant analysis.
```

| |
|---|
| To use the SDK in this mode, use one of the `EBUSDK::Analyzer::AnalyzeMXFFile()` functions that are documented in the Doxygen documentation. |

# 3 SDK Structure and Dependencies

In this section, a short description is given of the individual components of the SDK, and the limited set of software components it depends on, as shown in Figure 3. Basic MXF manipulation and parsing functionality has been inherited from the BBC's libMXF (the C library) and libMXF++ (a C++ wrapper library of the libMXF functions). libMXF++ (and as such also libMXF) are used by the BMX higher-level library for MXF manipulations for specific Application Specifications (e.g., AS-02, AS-11, …). This BMX library is in turn also used by the EBU SDK code, along with libMXF++ and libMXF. Separate code repositories are used for the libMXF, libMXF++ and BMX libraries and are contained in the SDK code as git submodules.

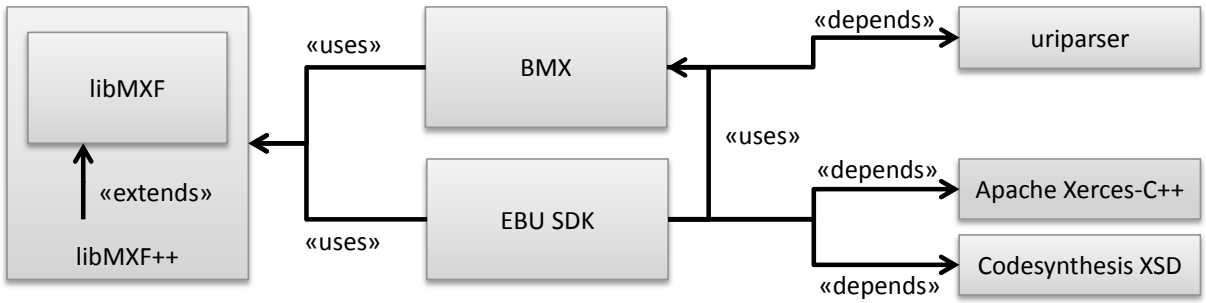


Figure 3: Dependencies in the EBU SDK.

Additionally, the SDK depends on the following software components.

- uriparser: a library for parsing URIs;
- Apache Xerces-C++: a library for processing XML documents and XML schema validation;
- Codesynthesis XSD: a library for generating XML schema-based parsers and serializers.

For this version of the SDK, programs use the SDK statically link to the functionality of the SDK, no explicit provisions have been made yet to produce a dynamically linkable library.

## 3.1 Code Structure

This section describes where in the repository the source code for each of the functional elements of the SDK can be found. For illustration, these elements have been identified in Figure 4.

| | |
|---|---|
| **ebu-mxfsdk\ebu-libmxf**<br>**ebu-mxfsdk\ebu-libmxfpp**<br>**ebu-mxfsdk\ebu-bmx**<br>**ebu-mxfsdk\uriparser** | Code repositories of SDK dependencies as git submodules. |
| ebu-mxfsdk\CustomMetadataSupport ↵<br>**\src\MXFCustomMetadata.cpp** | Contains generic code for handling custom metadata serialization and extraction in MXF files.<br>As such, it implements the generic part of the |

| | |
|---|---|
| | *(DE)MUX* block ❺ in Figure 4. This generic part also uses various functions and objects of the bmx and lower-level libMXF and libMXF++ libraries. |
| ebu-mxfsdk\CustomMetadataSupport ↵<br>**\src\XercesUtils.cpp** | Utility code for interaction between the Xerces-C++ functions and the functionality of MXFCustomMetadata.cpp, e.g., for the serialization of parsed XML documents into an MXF KLV packet. |
| **ebu-mxfsdk\EBUCoreProcessor** | Functionality specific to the EBUCore functionality of the SDK. |
| **ebu-mxfsdk\EBUCoreProcessor\apps** | Example applications that use functionality of the EBUCore functionality (and the bmx libraries). This includes the **raw2bmx, mxf2ebu** and **ebu2mxf** tools mentioned earlier. |
| ebu-mxfsdk\EBUCoreProcessor ↵<br>**\src\EBUCore_1_4\metadata***<br>ebu-mxfsdk \EBUCoreProcessor ↵<br>**\include\EBUCore_1_4\metadata*** | Contains the EBUCore source files that represent the EBUCore KLV object model for version 1.4 of EBUCore. This code is generated by the generate_ebucore_classes tool.<br>Also contains code (EBUCoreDMS++.cpp) that deals with registering the EBUCore metadata extensions with a standard MXF metadata model.<br><br>This implements the *KLV-based EBUCore object model* block ❹ in Figure 4. |
| ebu-mxfsdk\EBUCoreProcessor ↵<br>**\src\EBUCore_1_4\xsd***,<br>ebu-mxfsdk\EBUCoreProcessor ↵<br>**\include\EBUCore_1_4\xsd*** | Contains the source files that represent the C++ object model of the EBUCore XML schema for version 1.4 of EBUCore. This code is generated by the Codesynthesis XSD schema compiler. One file is present for each version of the XSD schema used (i.e., v1.3 and the latest version of 1.4 available at time of writing).<br><br>This code implements (along with the external XSD and Xerces-C++ libraries) the *EBUCore Parser/Writer* block ❶ and the *Schema-based EBCore object model* block ❷ in Figure 4. |
| ebu-mxfsdk\EBUCoreProcessor ↵<br>**\src\EBUCore_1_4\EBUCoreMapping.cpp**,<br>ebu-mxfsdk\EBUCoreProcessor ↵<br>**\src\EBUCore_1_4\EBUCoreReverseMapping.cpp** | Bi-directional mapping code for conversion between the XML Schema-based EBUCore object model and the KLV-based object model, for version 1.4 of EBUCore.<br><br>These files implement the *Bi-directional mapping* block ❸ in Figure 4. |
| ebu-mxfsdk\EBUCoreProcessor ↵<br>**\src\EBUCore_1_4\EBUCoreProcessor.cpp** | Contains EBUCore 1.4-specific code for handling EBUCore serialization and extraction in MXF files.<br>As such, it implements the EBUCore-specific part of the *(DE)MUX* block ❺ in Figure 4. |
| ebu-mxfsdk\EBUCoreProcessor ↵ | Contains EBUCore version-agnostic code for |

| | |
|---|---|
| **\src\EBUCoreProcessor.cpp** | handling EBUCore serialization and extraction. This class implements the developers interface of the EBUCore functionality and dispatches calls to subclasses for operations that are version-specific. |
| **ebu-mxfsdk\Analyzer** | Functionality specific to the Analyzer functionality of the SDK. |
| ebu-mxfsdk\Analyzer ↵<br>**\src\Analyzer.cpp** | Contains code for the generation of ST-434 reports from MXF file structure and metadata. This code implements the Analyzer block ❸ in Figure 5.<br>This code reuses KLV metadata definitions of the generic MXF structural metadata and EBUCore metadata in blocks ❶, ❷ in Figure 5. Additional metadata definitions (e.g., the AS-11 descriptive metadata), as depicted in block ❹ in Figure 5, can also be analyzed when additional metadata definitions are registered with the analyser, in the `EBUSDK::Analyzer::` ↵ `RegisterAnalyzerExtensions()` function. |
| ebu-mxfsdk\Analyzer ↵<br>**\apps\Analyzer.cpp** | Example applications that use functionality of the Analyzer functionality. This includes the **mxfanalyzer** tool mentioned earlier. |

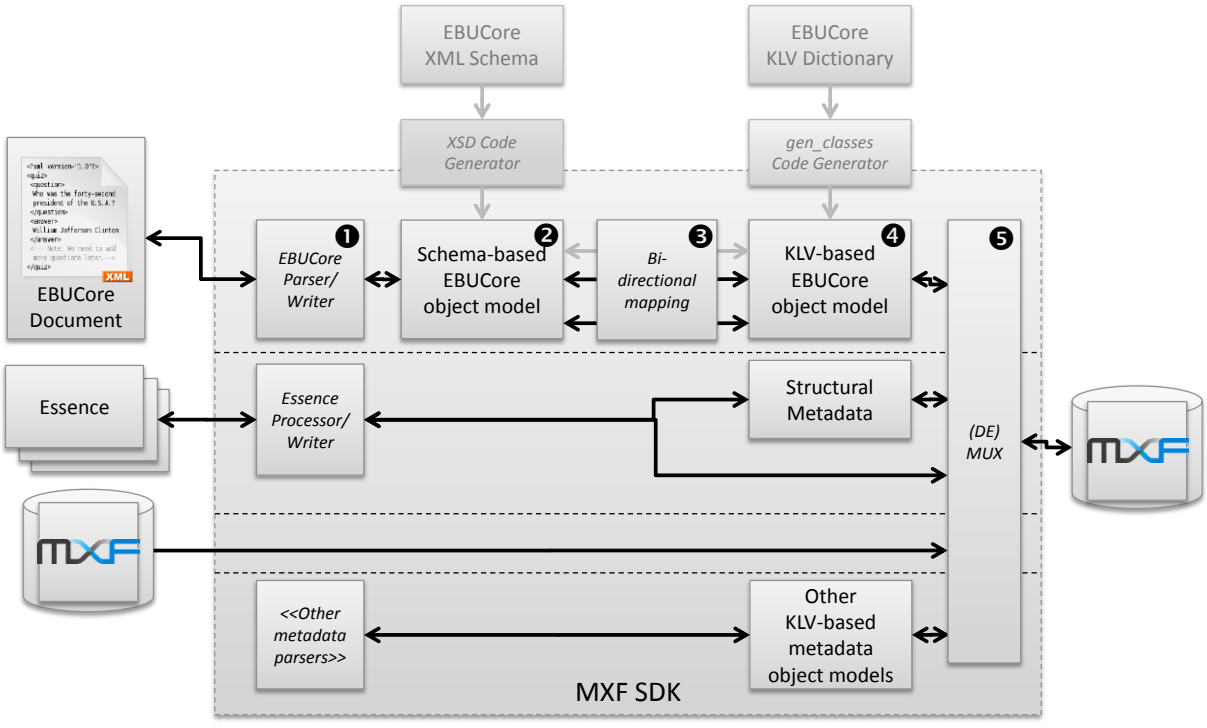


Figure 4: Functional code blocks of the EBUCore functionality of the MXF SDK.

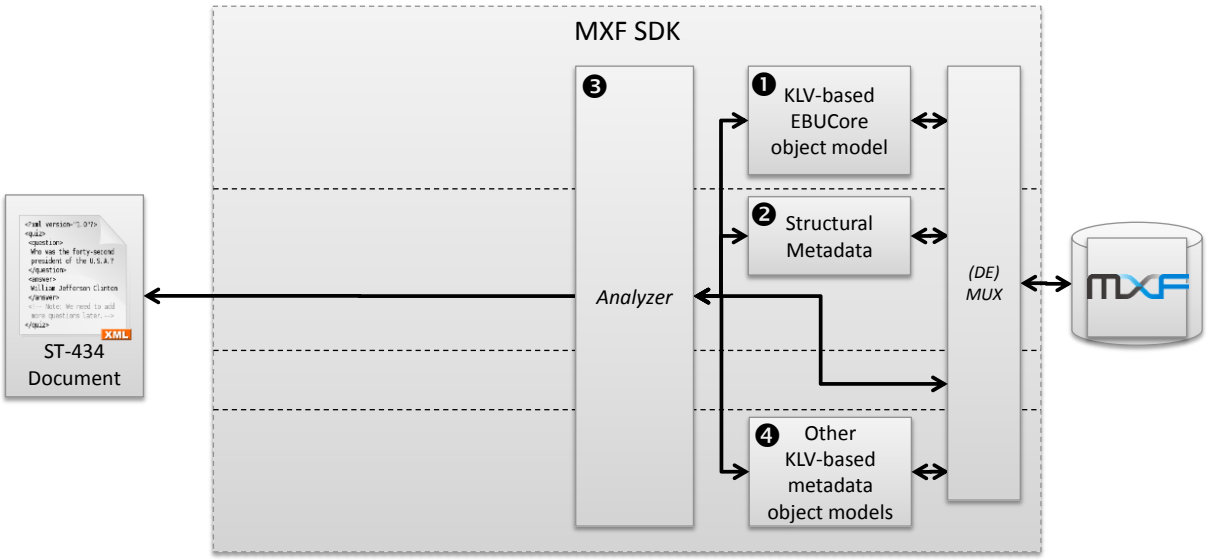


Figure 5: Functional code blocks of the analyzer functionality of the MXF SDK.

# 4 EBU SDK Source Code

The source code for the SDK consists of four repositories, cloned from the BMX libraries and related code by BBC. It is publicly available at the following locations:

| | |
|---|---|
| https://github.com/ebu/ebu-libmxf | Contains the base MXF functionality written in C, including code for reading KLV units, partitions, metadata, … |
| https://github.com/ebu/ebu-libmxfpp | C++ wrappers for the libmxf library, with a number of extensions for the EBU SDK. |
| https://github.com/ebu/ebu-bmx | Contains the high-level BMX libraries by BBC. |
| https://github.com/ebu/ebu-mxfsdk | Contains the MXF SDK source code in a git repository, with the above repositories linked in as submodules. |

### 4.1.1 Building the source code on UNIX-type systems

The SDK inherits the build system from the BBC BMX libraries, and uses a sequence of automatically generated configuration files that automate the build process, incl. tools such as GNU automake and GNU autoconf.

The code depends on a number of software frameworks that must be present before building.

| | |
|---|---|
| uriparser | This code is included as a git submodule of the SDK and will be checked out automatically. Then use the uriparser build instructions to make and install the library. |
| Codesynthesis XSD (version 3.3.0) | Depending on the system specifics, this component could be present in the system package/software management repository and can be installed from there. Otherwise binary packages and source archives are available at: http://www.codesynthesis.com.<br>Note: The EBU SDK requires only header files from the XSD component. No binary libraries need to be linked with SDK-produced software. |
| Apache Xerces-C++ (version 3.1.1) | Depending on the system specifics, this component is likely to be present in the system package/software management repository and can be installed from there. Otherwise binary packages and source archives are available at: http://xerces.apache.org/xerces-c/. |

The following commands build the SDK, and install the code it provides to a system-wide accessible location for other software to use.

Apart from the dependencies, this document assumes that all code repositories are checked out into a common parent directory from which the following commands should be executed.

```
cd libmxf
./gen_scm_version.sh
./autogen.sh
./configure
make
make install
```

```
cd libmxfpp
./gen_scm_version.sh
./autogen.sh
./configure
make
make install
```

```
cd ebu-bmx
./gen_scm_version.sh
./autogen.sh
./configure
make
make install
```

```
(from the ebu-mxfsdk directory)
./gen_scm_version.sh
./autogen.sh
./configure
make
make install
```

## 4.2 Building the source code on Windows systems

The source code includes Solution and Project files for the Microsoft Visual Studio 2010 development environment. From these project files, available in each repository's /msvc_build/vs10 directory, the project can be built directly, provided that the location of the include directories for the dependencies are updated or added in the Visual Studio projects.

The code depends on a number of software frameworks that must be present before building.

| | |
|---|---|
| uriparser | This code is included as a git submodule of the SDK and will be checked out automatically. Also, the Visual Studio project is included in the EBU MXF SDK solution. |
| Codesynthesis XSD (version 3.3.0) | The binary archive of this component can be found in the ebu-bmx repository: \dependencies\xsd-3.3.0-i686-windows.zip. This archive must be extracted [2].<br>Note: The EBU SDK requires only header files from the XSD component. No binary libraries need to be linked with SDK- |

| | |
|---|---|
| | produced software. |
| Apache Xerces-C++ (version 3.1.1) | The binary archive of this component can be found in the ebu-bmx repository: \dependencies\ xerces-c-3.1.1-x86-windows-vc-10.0.zip. This archive must be extracted [2]. |

[2] When the SDK has been installed using the installer, the extraction of the archive is performed automatically.

# 5 Appendix 1: Generating new EBUCore definition code

Whenever the definition of a metadata representation (the XML Schema or KLV registers) of EBUCore changes due to revisions of the standard parts of the code of the EBU SDK can be regenerated to reflect those changes. This Appendix describes how this is done.

Note that for the generation of some EBUCore program code, a working runtime of the Java language is required.

## 5.1 Generating new EBUCore XSD schema definition code

The EBUCore XML schema is compiled into C++ code that represents the model of the schema such that it can be manipulated using program code. Whenever the XML schema changes, the code must be regenerated using the Codesynthesis XSD schema compiler. A windows batch file is provided in the ebu-mxfsdk/EBUCoreProcessor/schema/compilexsd.bat to perform this operation. Update the location of the xsd.exe compiler in this file and then execute the batch script to generate updated source files (which are copied into the proper directory automatically).

Note that the compilexsd.bat file must be updated to refer to newer versions of EBUCore (and update the EBUCore schema name) if the schema would represent a newer version of EBUCore.

## 5.2 Generating new EBUCore KLV-based definition code

Whenever the definition of the KLV-based representation of EBUCore changes, the program code that implements it must also be regenerated, using a combination of tools provided with the EBU MXF SDK. The sequence of execution is illustrated in Figure 6.

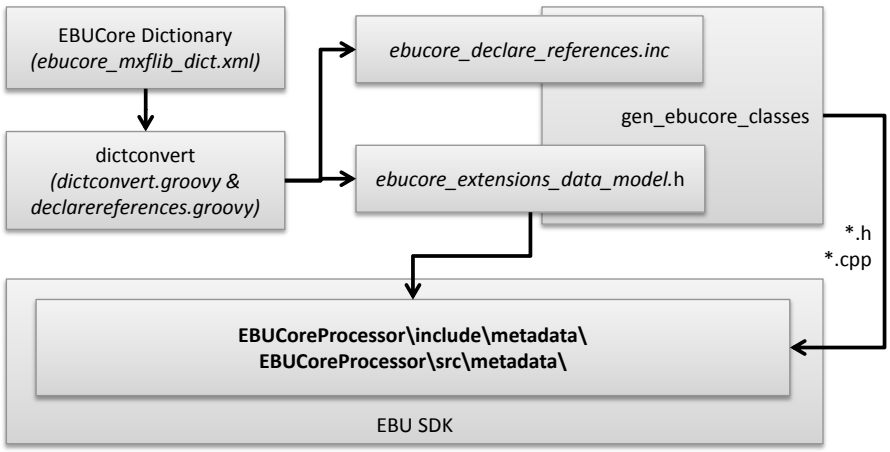


Figure 6: Sequence of KLV-based model code generation.

First, the dictionary (*EBUCoreProcessor/dictionary/ebucore_mxflib_dict.xml*) must be updated to reflect the changes of the KLV register specifications, which is a trivial operation (the existing elements in the dictionary serve as an exhaustive example).

The dictionary is then converted to a C header file (*ebucore_extensions_data_model.h*) which contains the low-level definitions of the EBUCore KLV-based model as an extension of the standard

MXF data model defined in the libMXF project. This conversion is done using a batch file (*EBUCoreProcessor/dictionary/dictconvert.bat)* and a number of java classes (*dictconvert.class* and *declarereferences.class*). Note that, if the name of the new EBUCore schema differs from the old one, the path to and name of included libraries pointing to the EBUCore schema (in the *EBUCoreMapping.h*) file need to be corrected. The *#include* directive must point to the new EBUCore schema generated by the compilexsd.bat batch file.

The generated header file (along with the *ebucore_declare_references.inc* file which defines how MXF metadata sets link with each other) is used by the *gen_ebucore_classes* in the libMXF++ project to create the metadata definition code. Recompile the gen_ebucore_classes project with the updated files and then execute it with the destination directory of where to create the source files as the single argument (make sure the destination is empty when doing so). The resulting C++ header and source files (along with the *ebucore_extensions_data_model.h* header file) can then be copied to the respective *EBUCoreProcessor/include/metadata* and *EBUCoreProcessor/src/metadata* directories where they can be employed by the EBU SDK.
Note that, the *gen_ebucore_classes* tool doesn't remove generated source files that are no longer contained in the model automatically. As such, if any of the metadata sets have been removed from the model, their corresponding source files must be deleted manually (or the *gen_ebucore_classes* tool must be setup to generate files in an empty directory).

When new classes are added to the metadata model, the *EBUCoreProcessor/include/<version>/metadata/EBUCoreDMS++.h* and *EBUCoreProcessor/src/<version>/metadata/EBUCoreDMS++.cpp* file should be updated, respectively with an #include directive for the newly created header file in the metadata directory and with a declaration of the object factory for the new object type (existing entries present in the file can serve as examples). Conversely, classes that are no longer in force must also be removed from these files. Additionally, the project files (or Make build system input files) must be updated to reflect the generated code changes; add or remove *.cpp* source files where applicable.

Finally, note that while the object model code can be generated for both the XML Schema and KLV-based representations, the bi-directional mapping between them (in EBUCoreProcessor\src\<version>\EBUCoreMapping.cpp and EBUCoreProcessor\src\<version>\EBUCoreReverseMapping.cpp) must still be updated manually, as following:

- Delete the mapping functions which reference deleted metadata sets. If a set definition wasn't deleted but upgraded or even renamed, its dedicated mapping function must be corrected to support new or updated fields. If mapping functions are no longer required, their references from other functions that generate objects of removed set types, must also be updated (e.g., when arrays or objects are created and mapped using the mapping function for each array elements).
- The declarations of mapping functions are not declared in a separate header file but immediately in the C++ mapping source files. As such, due to the nature of C++, functions must be defined before they are referenced in the source file.
- As is, the EBUCoreMapping.cpp file serves as an exhaustive example of the various mapping options, functions and macros to inspire developers in defining updated an new mapping functions.