2010-03-22

Withdrawal of EG 41-2004

Television – Material Exchange Format (MXF) – Engineering Guideline

> *A document should be Withdrawn only if there is a significant possibility of its use causing harm. A Withdrawn document shall still be made available and offered for sale by the Society, but it shall be prefaced by a cover page explaining its current status including a statement that some or all of the content is no longer endorsed by the Society.*

This Engineering Guideline has been withdrawn and its content is no longer endorsed by the Society. This action has been taken because it is judged that there is significant possibility that use of the document may cause harm.

The document no longer reflects the normative requirements of the referenced documents.

Changes to many normative parameters of various SMPTE MXF Standards and Recommended Practices have been made that render the information in this document incorrect or in conflict with the normative provisions of the standards or RP's that define MXF.

Readers of this document are cautioned that the information in the document may not be in compliance with the latest MXF SMPTE Standards and Recommended Practices.

EG 41-2004

SMPTE ENGINEERING GUIDELINE

for Television — Material Exchange Format (MXF) — Engineering Guideline (Informative)

Table of contents

1 Scope

This guideline gives an introduction to and the background for the material exchange format (MXF). This guideline describes the technology involved in the format, the names of the various elements within the format, and the way in which the format may be used within the real world applications.

Some parts of the descriptions within this guideline are generic to file formats, while other parts are specific to the material exchange format. There are descriptions of the object-oriented technology used within the MXF format, as well as a discussion of the metadata that may be used within the file. There are worked examples within this guideline to guide implementers and hence improve the interoperability of applications using different MXF implementations.

2 MXF document structure

The MXF specification is split into a number of separate parts in order to create a document structure that allows new applications to be covered in the future. These parts are:

Part 1 – Engineering Guideline – Informative (this document is SMPTE EG 41)
Part 2 – MXF File Format Specification – Normative (SMPTE 377M)
Part 3 – Operational Patterns – Normative (e.g., OP1a is SMPTE 378M)
Part 4 – MXF Descriptive Metadata Schemes – Normative (e.g., DMS-1 is SMPTE 380M)
Part 5 – Essence Containers – Normative (e.g., the MXF Generic Container is SMPTE 379M)
Part 5a – Mapping Essence and Metadata into the Essence Container – Normative (e.g., Mapping MPEG Streams into the Generic Container is SMPTE 381M)

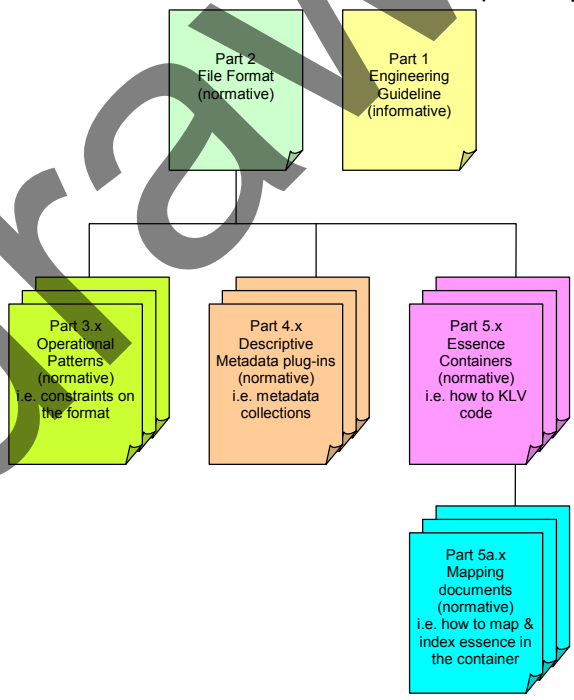
Approved
September 22, 2004

When implementing an MXF application or system, you should ensure that you have the latest version of all of these documents. The individual operational patterns and essence container mappings will be independently updated.

There are several parts to the MXF standard. This is Part 1, the MXF engineering guideline, which provides an introduction and description. This document should be read first because it introduces many of the concepts and explains what problem MXF is intended to solve. Part 1 also includes other engineering guidelines, including a descriptive metadata engineering guideline, which explains the concepts behind the use of descriptive metadata in MXF files.

Part 2 is a normative definition of the format of an MXF file. It is the toolbox from which different file interchange tools are chosen to fulfill the requirements of different applications. The MXF file format defines the syntax and semantics of MXF files.

Part 3 describes the operational patterns of the MXF format. In order to create an application to solve a particular interchange problem, some constraints and structural metadata definitions are required before SMPTE 377M can be used. An operational pattern defines those restrictions of the format that allow interoperability between applications of defined levels of complexity. Applications that use the MXF format must adhere to one of the operational patterns in order to achieve interchange.



Part 4 defines MXF descriptive metadata sets that may be plugged in to an MXF file. Different application environments will require different metadata sets to be carried by MXF. These collections of metadata sets are described in the part 4 document(s).

Part 5 defines the essence container of the MXF format for containing picture and sound essence. There may be limitations to the essence container that may be required in a particular operational pattern. The reader is advised to cross reference parts 3, 4 and 5 before and during implementation. The MXF generic container is a standardized essence container providing an encapsulation mechanism that allows many existing and future formats to be mapped into MXF.

Part 5a comprises a number of documents for mapping many of the essence and metadata formats used in the content creation industry into the defined MXF essence container.

The MXF document suite makes reference to other documents that contain information required for the implementation of an MXF system. One such document is the SMPTE dictionary, SMPTE RP 210, which contains definitions of parameters, their data types and their Keys when used in a KLV representation. Another is the SMPTE Labels Registry, RP 224, which contains a list of normalized labels that can be used in MXF sets. Annex B in this MXF engineering guideline contains a list of recommended string constants that an application may use to improve interoperability.

In the unlikely event of conflict or ambiguity between the different parts of the document, the format document has precedence over the operational patterns, which have precedence over the essence containers, which have precedence over the descriptive metadata documents.

NOTE – During the early development of MXF, a catalogue of enumerated values was created to list SMPTE labels, strings keys and tags used within the MXF document suite. The normative definition of the SMPTE labels is maintained in the SMPTE Labels Registry and the normative definitions of the SMPTE keys and tags are to be found in the MXF document suite.

2.1 About this document

The information in this document is ordered for the novice reader. Concepts are introduced gradually and repeated in more detail later in the document. This is done to make the document easier to read; however, it does make the document somewhat less good as a reference. For that reason, a table of contents is provided at the start of the document to allow “random access” to the information within the text.

Section 8 provides MXF worked examples. In order to improve the readability of the text, an arrow is used to indicate that an example of a certain subject exists for this section. For example, (🡻8.4) indicates an example for this subject exists in section 8.4.

3 Introduction

The introduction is constructed as a list of questions. The concepts in MXF can be introduced in a way that gives an overall view of the specification and the concepts embodied within it. Once the introduction is understood, the requirements of the file format are discussed. Some specific words and phrases used in the specification are then defined and finally the material exchange format is introduced in a much more detailed fashion. Although this entire document is informative, it is hoped that it will give sufficient information for technical and non-technical readers to understand MXF.

3.1 What problem is the material exchange format trying to solve?

The MXF specification is intended to encourage an environment where it is convenient to interchange multimedia information as a file. This will allow users to take advantage of non-real time transfers and to package together essence and metadata for effective interchange between servers and between businesses. MXF is not a panacea, but is an aid to automation and machine-machine communication. It allows essence and metadata transfer without the metadata elements having to be manually re-entered.

The MXF specification is intended to allow the interchange of captured, ingested, finished or “almost finished” material. It is not intended to be an authoring format. Despite this, careful thought has gone into SMPTE 377M to ensure that authoring tools such as those based on AAF Association technology are able to directly open and use an MXF file efficiently without having to convert the file.

The MXF specification has also been carefully crafted to ensure that it can be efficiently stored on a variety of media, as well as transported over communications links. The MXF format has not forgotten about tape. There are structures and mechanisms within the file that make MXF appropriate for data tape storage and archiving of content.

Finally, the MXF specification is intended to be expandable. A considerable effort has been put into making SMPTE 377M compression format independent, resolution independent and can be constrained to suit a large number of application environments. The document structure has been created to allow new applications to take advantage of the MXF format in a backwards compatible way.

3.2 How does MXF satisfy the design requirements?

3.2.1 Basic Structure

The MXF format follows a common theme of many file formats and has the following basic structure:

A file **header** that provides information about the file as a whole, including labels for the early determination of decoder compliance.

A file **body** that comprises picture, sound and data essence stored in essence containers (see 3.5.8). Essence containers from different tracks may be interleaved or separated. The section on operational patterns goes into more detail on this subject

A file **footer** that terminates the file. The file footer may include some information not available at the time of writing the header (such as the duration of the file). In certain specialized operational patterns, the file footer may be omitted.

A simple MXF file is shown in figure 1.

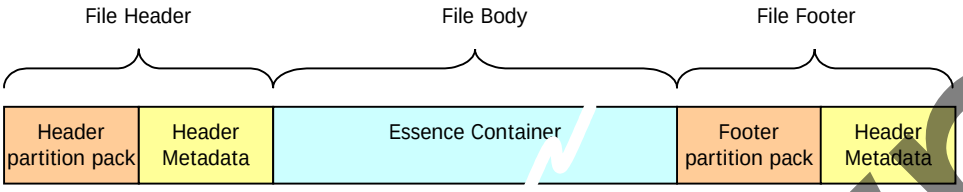


Figure 1 – Simple MXF file

MXF files may include an optional, but recommended, **index table** that provides rapid conversion from sample-based indexes (e.g., time code) into byte offsets within an essence container. The index table may be segmented, and may be stored before, after or multiplexed with the essence data segments.

MXF files may also include optional file body **partition**s that can be inserted at intervals within the File Body and are used to provide a variety of features:

1. Robustness of metadata information by repetition of the header metadata;
2. Multiplexing of different essence containers;
3. Distributing an index tables in small chunks (e.g., for devices with limited memory);
4. Providing “per-stream” index tables that are position independent within the file;
5. Easier location of essence container data when using high speed tape devices;
6. Optimizing the distribution of the data in a file for storage or transmission.

Repetition of the header metadata within a body partition is dependent upon the application on a per-application basis. Such applications are to be found in the transfer of an MXF file as a stream over a unidirectional link and in data tape shuttling. One purpose of such header metadata repetition is to support the recovery of critical metadata in applications where the file may be interrupted or where the decoder starts to receive data in mid-transfer.

Multiplexing and storage optimization is a complex subject and is highly dependent on the storage or transmission device used. Hard discs, DVDs, satellite links and tape devices all have different requirements. The MXF structure allows a great deal of flexibility in the positioning of the partitioning information and the use of fillers to allow optimization for different devices. Typically, if storage or transmission optimization is important in an application then the MXF encoder will know which parameters are important to it. MXF provides the tools, but encoders can make the optimizations that add value to their implementations.

MXF files use key-length-value (KLV) coding throughout for flexibility and extensibility. KLV coding is defined in SMPTE 336M; a full review was published in the July 2000 edition of the SMPTE Journal (Vol. 109, No 7, Engineering Report). This mechanism is used to encapsulate the individual elements of an MXF file in such a way that devices can ignore information when the key of a KLV triplet is unknown. The length parameter tells the KLV decoder how much data should be ignored.

In specialized operational patterns, the header (see section 3.5.2) is allowed to start with a non-KLV run-in. This is to allow synchronization bytes or “camouflage” bytes to be added at the front of the file in certain (limited) applications. In all other circumstances, there will be no run-in and the entire file must consist of only of KLV elements with NO gaps.

3.3 Two ways of viewing an MXF file

An MXF file can be viewed in two ways:

There is the **physical** view of the MXF byte stream on disk or on the wire.

There is the description of the file contents obtained by decoding the data model. This will be referred to as the **logical** view of the file.

These two views are summarized in figure 2.

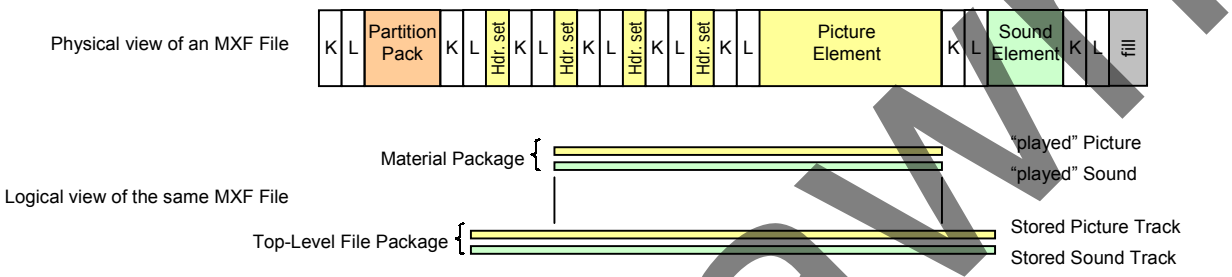


Figure 2 – Physical and logical views of an MXF file

3.3.1 Physical view of an MXF file

This is the simplest way to view the file. Many MXF processes and applications will use this layer only. Some of the physical properties of the file are the partitions, the KLV coding, the index tables, the run-in, the KLV alignment grid (KAG) and the random index pack (RIP).

The physical properties of the file are largely independent of the number of tracks in the file, the amount of metadata carried and the relationship between the different picture and sound elements. The way in which an MXF file is written is MXF encoder and application dependant. Many application specific optimizations may be incorporated into an application to improve the way an MXF file is physically written to a device.

- Physical optimizations may include any or all of the following:
- Matching the KLV alignment grid (KAG) to an integer multiple of the underlying physical sector / cluster / packet size of the medium;
- Adding body partitions with repeated header metadata to allow recovery from an interrupted transmission;
- Using the run-in mechanism to camouflage the MXF file as a different file type;
- Repeating index tables in the file header and file footer for easy access in a tape environment;
- Adding a random index pack to quickly find all the partitions in a large file.

3.3.2 Logical (metadata) view of an MXF file

The logical view of the file is defined by the contents of the MXF metadata and not by the way in which it is organized as a byte stream. The metadata defines the number of different picture, sound and data tracks as well as the descriptions of the different essence types within the file. Figure 2 shows a very simple MXF file that contains a material package and a top-level file package, each of which has a single picture track and a single sound track. The data is physically stored in KLV coded triplets and organized by partitions as shown in the upper portion of the diagram. The lower part of the diagram shows what the metadata in the file is **intended to represent**. Bear in mind that this logical representation is very compact — a typical file package will be less than 1 kbyte, whereas the essence it represents may be megabytes, gigabytes or even terabytes.

The material package can generally be thought of as the “output timeline” of the file. The top-level file package can be thought of as the stored data or “input timeline” of the file. The metadata within the file describes the stored data within the file as well as the portion that is to be output when the file is played or used in some way. The example in figure 2 shows that all the tracks of the stored data (in the top-level file package) are used in the material package, but an MXF player will play only a small segment from the middle of the file.

3.3.2.1 Structural metadata

The structural metadata is the way in which MXF describes different essence types and their relationship along a timeline. The MXF structural metadata defines the way in which the output timeline of the file relates to the one or more stored top-level file packages. The structural metadata defines the synchronization of different tracks along a timeline. It also defines the picture size, picture rate, aspect ratio, audio sampling and other essence description parameters.

The structural metadata is defined in SMPTE 377M. Most of the parameters are defined in the MXF file format document, but additional descriptors and labels may be defined in essence mapping documents. The MXF structural metadata is derived from the AAF data model. This means the relationships between all the different sets and their properties are precisely defined. More information on the structural concepts appear later in this document.

3.3.2.2 Descriptive metadata

MXF descriptive metadata comprises information in addition to the structure of the MXF File. This may be intended for human use (as in the majority of the SMPTE 380M: MXF DMS-1 specification) or it may be information for machine use, such as a track of information containing depth information for 3D processing. SMPTE 377M provides a very simple plug-in mechanism that allows different Metadata sets to be defined and used in an MXF environment. SMPTE 377M provides mechanisms for uniquely identifying the metadata scheme(s) present in the file, mechanisms for preventing numerical conflict with existing metadata and a mechanism for determining the version of the descriptive metadata specification used.

The MXF metadata plug-in scheme was developed as a result of strong user requirements. No single metadata definition and structure will be appropriate for everyone. A mechanism that properly allows the integration of new metadata schemes without redeveloping applications and equipment needed to be created. The MXF plug-in mechanism is very lightweight and allows versatility for the implementers and extensibility for the users.

When descriptive metadata is added using the plug-in mechanism, many of the features of MXF are achieved automatically. The ability to create multiple tracks and synchronize them against each other, the ability to add metadata events synchronized with the video / audio or other tracks and the ability to use metadata in the output timeline that was available in the source file are all part of the standard MXF feature set. This document will outline only the basics of a descriptive metadata scheme. A fuller treatment of the subject can be found in the descriptive metadata engineering guideline, SMPTE EG 42. It is worth noting that descriptive metadata can be for both human and machine use. Much of the machine-descriptive metadata relates to special properties of the essence and has an intimate spatio-temporal relationship to the essence. For this reason it is often called Intimate metadata.

3.3.2.3 Dark metadata

Dark metadata is the term given to metadata that is unknown by an application. This metadata may be privately defined and generated, it may be new properties added to SMPTE 377M or it may be metadata that is part of the MXF standard, but not relevant to the application processing the MXF file. It is important that there are some rules on the use of dark metadata to prevent numerical or namespace clashes when private metadata is added to a file that already contains dark metadata. Rules are given in the SMPTE 377M along

with the specification of a data structure called the primer pack. Guidance on the use of this structure is given in section 8.5.1 of this document. (🡻8.5.1)

3.4 What is the header metadata?

Although only occupying a small fraction of the size of a typical MXF file, the header metadata is often, for those inexperienced in data models, the most difficult part to understand. The following sections introduce the topic of object-oriented coding in a general and easy to understand manner. For a more rigorous explanation, there are many reference books that cover the principles and methods of implementation in far more detail than given here.

3.4.1 Why an object-oriented approach?

The underlying data structure of MXF was chosen to be a subset of the AAF data model. This AAF data model uses an object-oriented approach so MXF adopted the principle. This document will give a brief outline of some of the concepts. More detail on how this relates to the descriptive metadata structure is given in SMPTE EG 42.

3.4.2 But what is an object-oriented approach?

This is a technique for describing the functionality of a complex system by describing each of its components as though each is an independent object (or thingy or blob – whatever word is easiest). The quickest way of explaining objects is by means of an example. We will use the **track** object.

A track can be thought of as a straight line on a piece of paper. It starts at the start; it ends at the end and it lasts for its duration. The start, end and duration are known as properties.

A feature of object orientation is “inheritance”. This means that we can have different sorts of track. They all share some common properties that they inherit from the parent or super class, but have extra properties or functionality added to make them useful. For example, consider an **event track**. The straight line on the piece of paper can now be marked with events. An event can start at any point along the track. It may be instantaneous (i.e. no duration) or it may last for a defined time. Events may also overlap.

Another sort of track is a **timeline track**. Similar to an event track, it starts at a certain time, ends at a certain time and has a duration. This track has a restricted functionality in that it only allows source clips to be placed on the track. All the source clips must be contiguous, which means there are no overlaps and no gaps. Both of these track types inherit properties and functionality from a common track class.

This principle is the basis for the object-oriented definition of the MXF file format. The definition of the classes from which MXF objects are created comes primarily from the AAF Association class model. Generic classes with general functionality are defined. Classes with specific functionality then inherit the general class features. SMPTE 377M restricts some of the flexibility of these AAF classes to define the MXF sets. MXF applications populate these sets with values to create MXF objects in a file.

During the development of MXF, a zero divergence doctrine (ZDD) was created in order to ensure that any change in the model of behavior between AAF and MXF was severely restricted and eliminated wherever possible.

3.4.3 What sort of metadata can be put in?

Broadly speaking, the metadata items can be split into two groups: structural metadata and descriptive metadata as described above. The structural metadata is intended to bind the different elements of the file together and is needed to define the basic file structure. The descriptive metadata is intended to supply extra information about the file such as a program name or scene description. There are a large number of metadata elements defined in the SMPTE dictionary and in SMPTE 377M. To understand the restrictions on the use of metadata elements, it is necessary to understand the terminology in section 5.2.

3.4.4 Where does all the metadata go?

It depends on the metadata. The MXF object model creates “hooks” on which the metadata can be placed. These hooks live in the file header, the body partitions, some essence containers and the file footer.

The header metadata area is able to contain descriptive metadata that allows a production to be described. For example production, clip and scene information is described in the MXF descriptive metadata scheme 1 document (part 4 of this specification).

There are certain metadata parameters that might live in multiple places. The most obvious of these is time code. This may exist in the header metadata, but might also live embedded within the essence container data; e.g., in the GOP header of an MPEG essence container. This repetition is often important and the handling of any conflict between the different instances of the data is application dependent.

3.4.5 How does AAF fit into the big picture?

At a first glimpse, the relationship is obvious; MXF is for simple transfers and AAF is for authoring. Both formats exist to aid interchange of program material as files, which in turn will increase interoperability between file-based products.

The meaning of the opening sentence is a little more difficult than it first seems. "Authoring" can be seen as a catch-all phrase for a series of complex processes that take pieces of video and audio essence and put them together using a variety of composition effects (cuts, dissolves, DVE, rendering, magic). When the authoring process is complete, the "finished" program material can then be exchanged as a file. This is a simple transfer of the compiled / rendered / etc., program.

The complexity of AAF has been simplified so that we can state that: "*MXF files apply a subset of the AAF class model*". This means that the complexity of the authoring file format has been simplified. However, beware; "simplified" does not mean "completely obvious and like SDI". It is important to remember here that we are mixing two very complex and different worlds – A/V and IT.

When video engineers look at a series of words in an SDI stream, there is an implicit understanding of the complex spatio-temporal sampling and visual processing that went into creating those data words. A video engineer would take great care before modifying any value to ensure proper clipping, filtering and possible gamma correction took place.

The IT environment that has created AAF is just as complex (and just as "obvious" to those practiced in the art). AAF arranges its file format in terms of objects. These objects are chosen and defined to reflect the actual processes and content items that go into the authoring process. AAF is so powerful that the physical representation of these objects could be redefined providing no information is created or lost. An IT engineer would take great care before modifying the object model to remove things that looked like they were not needed - the implications for future enhancement and interoperability might be very serious and not "obvious".

“MXF files apply a subset of the AAF class model," means that MXF contains just enough of the AAF object model to allow it to represent a file interchange. This means it can represent an output timeline that has video, audio and data. It has a logical metadata structure, a defined physical representation (KLV) and is interoperable with other MXF systems and upward compatible with AAF. It has been designed so that an AAF system can open an MXF file without modification to either the MXF file or the AAF System.

In practical situations this means that there is a lot of overlap between MXF and AAF functionality. MXF is targeted at interchange throughout the broadcast and content creation chain, whereas AAF is optimized for round-tripping in Post-Production. As a rough rule of thumb, content interchange, cut-edit functionality or simpler is an MXF application; AAF is more appropriate for everything else. More details are given in annex A.1.

3.4.6 How do we represent time and timelines with tracks?

Time and timelines are features of the **logical** representation of an MXF file. The concept of time within the file is independent of the arrangement of bytes within the file, although constraints may be applied in order to get certain functionality from the file (streaming ⬇8.2.1). Time is used to measure the duration of the content as well as to synchronize the content. In MXF a "track" is used to represent the passage of time. A track has units to represent time and has an associated duration property. Some tracks have segments that butt against each other to form a continuous sequence of video (timeline tracks) whereas others may have overlapping events that refer to the point at which descriptive metadata is valid (event tracks). In fact, the mechanism for adding new descriptive metadata definitions to an MXF file is to add new tracks on which to "hang" the metadata items.

To synchronize two tracks, they must be somehow related. This is done by putting them within a package (a container for tracks) that synchronizes the start and duration of multiple tracks. Note, however, that the tracks may have different time measurement units within the package. Time is normalized within a track by its "edit rate" property. This in turn gives us an edit unit, of *1/edit rate*.

3.4.7 What are the units of time?

There are two main units of time used with an MXF file. These are:

Edit Units = *1/Edit Rate* – used to mark time along a track
Sample Units = *1/Sample Rate* – used to describe the underlying sample rate of the essence

Edit units may be chosen for the convenience of the file writer, whereas sample units define the sampling structure of the essence. A sequence of audio samples, for example may have a sample rate of 48 kHz, whereas the track that describes the sequence may have an edit rate of 50 Hz so that synchronization with parallel tracks is numerically simplified. For video streams, the sample rate is usually defined as the field or frame rate of the content and not the sampling clock.

3.5 How does MXF manage the complexity?

An MXF file is highly structured. There are different structural elements that divide the file in different ways to make the complexity manageable. This section describes some of these structural elements along with the reasons for the division.

3.5.1 What are the file header, the file body and the file footer?

The basic file header, file body and file footer are explained in section 3.2.1. The reason for the split is quite simple. The file header is designed to be small enough that it can easily be isolated and sent to a microprocessor for parsing. The bulk of the file will usually be the file body — this is the picture, sound and data essence. The file footer provides a means to put the header metadata at the end of the file. Why? In certain applications such as recording a stream to an MXF file, there will be header metadata values that won't be known until the recording is finished. The file footer provides a mechanism for doing this. It also provides clear indication that the file has terminated.

3.5.2 What is a partition?

A partition is a division of data within the file. There are three different sorts of partition, each of which can have four states:

Header Partition – this is the first partition of the file;
Footer Partition – this is the last partition in the file;
Body Partitions – all the other partitions are in the middle of the file and are used to divide the essence container(s) in a certain way.

A partition may be open or closed, except for the footer, which may only be closed. The normative definition of these terms is in SMPTE 377M and extra clarification is given here:

Open – This marks the information in a partition with a "caution" notice. Any metadata information in the partition was correct at the time of writing, but the application writing the file had not completed the writing process. This means that some of the information may be absent, or may turn out to be plain wrong when the file is ultimately closed. For example, a capture device may have identified a picture and a sound track when it initially started writing the file. During the writing process, a second sound track commenced — this track was not described in the open header metadata.

Closed – This marks any metadata information in the partition as finalized. The application or device creating the file correctly terminated the file and all the properties of the Metadata sets were filled in to the best of the application's ability. In the example above, a repetition of the Header Metadata would be placed in the footer that correctly described the existence and duration of the second Sound Track. All closed partitions in a file must have the same metadata property values. This is mandatory. This allows an MXF decoder to determine that the metadata is correct as soon as it finds a closed partition. SMPTE 377M states that the file footer, if present, will always be a closed partition.

An MXF File can only be called a "closed" file if there is at least one closed partition with metadata. It is important to note that robustness is enhanced when all the partitions in a file are closed (🡻8.2.6). If a file is accidentally truncated during a transfer and the only closed partition in the file was the footer, then the file is no longer a "closed file". If robustness is desired (and it usually is), application and device developers are urged to close all the partitions of their files. All valid MXF files must be closed however certain situations, such as an interrupted file transfer, may leave an "open" file that is still partly usable. The ability of a device to handle "open" MXF files is an application issue.
In an ideal world, the two states of "open" and "closed" would be sufficient to describe all the files in existence. The desire for cheap hardware and software, however, means that some capture devices and applications will not be able to parse the wide variety of essence types they might expect to place in an MXF file. To cope with this condition, the states "complete" and "incomplete" have been defined to mark the status of the essence descriptor(s) in the MXF file.

Complete – Each of the properties in the header metadata with a status of "required" or "best effort" exist in the file and are correct. The status of each of the properties is given in SMPTE 377M.

Incomplete – One or more properties within in the header metadata with a status of "best effort" has a distinguished value. The distinguished value is used to mark the property as "unknown at the time of writing". An MXF file may still be a closed file because all the other properties of the file are known. Some of the header metadata may be incomplete due to the absence of an essence parser at the time of file creation. This allows an application to report many of the metadata properties of the file, but certain essence decoders may need to parse portions of the file before it is playable.

Maximum robustness is achieved when applications and devices create Closed and Complete MXF Files. (🡻8.2.6).

Each partition starts with a partition pack that defines what sort of partition it is, followed by the following optional items:

- Header metadata
- Index table segment(s)
- Essence container data

From these and other restrictions, we limit an MXF partition to contain only a single "thing"; i.e., a single essence type with its associated index table segments. If different essence containers need to be multiplexed together within the file, then a new partition must be started when the essence container changes.

3.5.3 How does KLV leave room for expansion?

At a lower level than the object definitions is the KLV coding. KLV stands for key-length-value. Every object, piece of metadata or any "thing" in the MXF file has a **key** (16-byte value) and a length that defines how long the value of the object, metadata or "thing" is. After this, the value of the object, metadata or "thing" follows. Note that the key is in fact a SMPTE Universal label and as such follows the rules defined in ANSI/SMPTE 298M. KLV coding is fully defined in SMPTE 336M and includes not just the encapsulation of individual data items, but also the encapsulation of collections of individually coded KLV data items into logical data sets and packs (a.k.a. objects as above).

A decoder that does not recognize a key is able to skip over the unknown value and inspect the next key. This allows extra functionality to be added to the MXF specification at a later date, knowing that older decoders will be able skip over the values.

Words within the key are ISO object identifiers (OID) using primitive BER (basic encoding rules: ISO/IEC 8825-1 ASN.1). This means that the most significant bit of each 8 bit value is a flag to say that the word is greater than a 7 bit value. For example if the 12 bit value **b** (b_{11} .. $b_{0)}$ is to be mapped into a KLV key then here is a possible mapping into bytes 14 and 15 of a key:

Word. bit	**14. 7**	**14. 6**	**14. 5**	**14. 4**	**14. 3**	**14. 2**	**14. 1**	**14. 0**	**15. 7**	**15. 6**	**15. 5**	**15. 4**	**15. 3**	**15. 2**	**15. 1**	**15. 0**
value	1	0	0	b_{11}	b_{10}	b_9	b_8	b_7	0	b_6	b_5	b_4	b_3	b_2	b_1	b_0

Figure 3 – Example of BER OID encoding

Figure 3 shows a binary 1 in bit 7 of byte 14 to indicate that this is a multi-byte value. There is a binary 0 in bit 7 of byte 15 to show that this is the last byte of a multi-byte value. A byte value of 0 is often used to terminate a label and a marker bit in bit 6 of byte 15 may be used to prevent accidental termination from occurring. Note that the actual mapping of bits into a label Key must be normatively defined in an appropriate document.

NOTE – At the time of writing this guideline, this multi-byte OID technique is not in use in any of the specifications. MXF parser writers should be aware that this technique may be used in the future, and that, although the number of bytes in a SMPTE key is 16. The number of words may be less than 16, or alternatively, there may be 16 bytes of which the final words are assumed to be 0.

The **Length** field is BER (Basic Encoding Rules: ISO/IEC 8825-1 ASN.1) coded. This allows the length field to have a variable number of bytes. So how do you know the length of the length field?

The length field is always coded MSB (most significant byte) first. If bit 7 of the first byte is a '0' then the 7 least significant bits contains the length value (0 .. 127). If bit 7 of the first byte is a '1' then the 7 least significant bits tell you the number of bytes in the length field; e.g., the value '83_h' means that the next 3 bytes contain the length field. The format document gives recommendations for the upper limit of the length field. Decoders must be able to handle both long form and short form BER coding.

The examples below show a length value of 64 coded in the three different ways:

40_h	short form coded
83.00.00.40	long form coding using 4 bytes overall
87.00.00.00.00.00.00.40	long form coding using 8 bytes overall

3.5.4 What is a KAG?

It is a KLV alignment grid. This is a performance enhancer for devices with fixed size blocks. During the design of the MXF format there were many discussions on whether the format should use rigid sectoring or

not. The conclusion was that sometimes it was important, but a device or application should always be able to read an MXF file regardless of whether the elements within the file fell on rigid byte boundaries within the file.

The KAG can be thought of as gridlines spaced on uniform byte boundaries in each partition. To achieve good performance, all the important KLV items within the file (header metadata, content packages of the essence, etc.) should line up on the grid. This means that the first byte of the key should be on a grid boundary.

The reference point for a KAG is the first byte of the key of a partition pack, and the KAG value is valid within the partition. SMPTE 377M states "*The first gridline in any partition is the first byte of the key of the partition pack that defines that partition.*". In order to have a global KAG value, each and every partition pack must have the same KAG value. Additionally, to maintain this global KAG value, the first byte of each and every partition pack must lie on a KAG boundary. Finally, if there is a run-in, its length in bytes must be an integer multiple of the KAG value.

This feature is a performance enhancer because it reduces the need to search every byte for the start of a new file component. It is possible that some process may make a change to a file that breaks the KAG rules, but is unable to modify the KAG value in the partition header. An MXF decoder that is receiving a file may desire a certain KAG value because its internal storage is arranged on rigid boundaries. It should continue to check each of the KLV triplets received for confirmation that they still lie on the KAG. The majority of files that use the KAG feature will respect the value in the partition header, but some may not. The MXF application receiving the file that does not respect the KAG should not fail under this condition, but performance may be severely restricted. For example, the receiving application may choose to process the incoming stream to force it to be aligned to the KAG by inserting fill KLVs. This may slow it down and cause it to recalculate index tables.

3.5.5 What is an index table?

An index table improves random access within an MXF file. Specifically, it allows random access by a time index. This means that if you want to access the picture, sound or data that starts 10 seconds into the file, then an index table will provide the translation between the time value and the byte offset within the file. MXF index tables are quite complex because the format is designed to cope with interleaved essence containers that may be constant or variable bit rate and that also may be temporally reordered on the disk compared to the presentation order (e.g., long GOP MPEG-2 files). Index tables are more fully described in section 8.3.

3.5.6 What is a random index pack?

A random index pack (RIP) provides a list of the positions of all the partitions within a file. This is different from an index table, which provides the byte offsets of the content within a partition. The difference can be clearly seen when two different essence containers are multiplexed together. There will be two separate index tables, each of which contains conversions between temporal offsets and byte offsets within each essence container.

The RIP, however, gives absolute positions of the partitions, so all the index tables may be rapidly built without parsing the entire file. The RIP contains a mechanism for quickly determining its existence.

3.5.7 What is an operational pattern?

An operational pattern is used to constrain MXF complexity. The generalized operational patterns are intended to split the complexity depending on the complexity of processing required by an MXF decoder. Specialized operational patterns are likely to be created in order to constrain MXF for a particular "application space". Usually an MXF file is interchanged for a purpose. This may be the exchange of an ingested clip, a camera output, a finished program or the interchange of a partially edited program. Both of these requirements have different implications for the structure of the file. Different operational patterns define the structural metadata that is required to satisfy a particular application. In general, the higher the number of the operational pattern, the more complex the file and the more functionality is required in the decoder. Simple

operational patterns such as OP1a can be used with both linear and non-linear access devices. Some more complex operational patterns require non-linear access devices.

Each operational pattern has an assigned SMPTE label value that allows MXF decoders to quickly recognize the complexity of an MXF file.

3.5.8 What is an essence container?

An essence container defines the encapsulation of a particular type of essence. Its purpose is to allow the essence to be wrapped in KLV and to have associated with it an optional index table to allow rapid access to a given time offset within the essence. The essence container is structured to allow easy multiplexing with other essence containers and to allow identification of the decoding requirements needed to display / listen to / play / execute the content.

An essence container specification defines a unique SMPTE label for identification as well as a method for encapsulating the essence in a KLV structure. Different essence containers may place restrictions on the interleaving of the essence data to be compatible with existing applications. The SMPTE label allows decoders to make a fast go/no-go check of the essence type at the very beginning of the file.

An MXF file may have more than one essence container. The precise number of essence containers and their relationships is constrained by the operational pattern with which the file complies.

A “generic container” is defined within MXF. This is intended to carry all the mainstream essence types in existence at the time of creating SMPTE 377M. It is very simple in operation, yet flexible enough to carry uncompressed material as well as reordered MPEG compressed material. Associated with the generic container are a number of mapping documents that define how the actual essence byte stream should be placed in the essence container.

3.5.9 How have we specified the essence container?

There are several specific questions that need to be asked when putting an essence container into an MXF file. These are notably:

1. **What limitations are placed on the essence container when it is in an MXF file?**
2. **Are there interleaved variants of the essence container?**
3. **How do we KLV code the contents of the essence container?**
4. **How do we pad the essence containers to fit the chosen KAG size?**
5. **What do we do with the metadata embedded within the essence container?**
6. **How do we use index tables with the essence container?**

The essence container and mapping specifications are basically recommended answers to these questions. It is the intention of the essence container and mapping documents to restrict the choices of an essence container implementation sufficiently to allow interoperability between devices, yet with enough flexibility to solve real world problems.

3.6 How does MXF interoperate with stream interfaces?

MXF files may be directly created from standardized formats such as MPEG-2 system and elementary streams, AES3 data streams and DV DIF packet streams. These formats may be mapped from one of several real-time interfaces such as SMPTE 259M (SDI), SMPTE 305.2M (SDTI), SMPTE 292M (HD-SDI), or transport interfaces with real-time protocols such as IEEE-1394, ATM, IEEE802 (ethernet), ANSI fiber channel and so on.

When a streaming file is captured, a file header is created and the essence is KLV wrapped on the fly. The data rate increases due to the KLV wrapping and addition of headers. Real time streaming devices must ensure that any buffering requirements of a streaming interface are catered for with this change of data rate.

Conversion to and from the source format is always possible, but sometimes there will be loss of information. Not all streaming and storage formats are able to store the rich metadata constructs available in an MXF file. Often there will be a lossy data mapping where information in one format cannot be represented in the other. Eliminating this undesired loss is a function of the systems engineering that interconnects MXF and non-MXF systems. In many formats such as the MPEG-2 transport stream, research is being done to find ways in which MXF headers can be "tunneled" through the transport stream so that its use in an MXF system provides transparency as well as interoperability.

3.7 How does MXF interoperate with other files?

As previously stated, MXF files apply a subset of the AAF class model. The material exchange format provides a data structure together with a set of constraints and plug-ins to create files that can be directly written and read by AAF systems. MXF is also able to inter-operate with other existing file formats by utilizing techniques such as external essence and using the run-in to "camouflage" the appearance of the file (see the end of section 3.2.1). Different metadata models can be plugged into the MXF file format to provide extensions and the KLV structure itself can be converted to formats such as XML for exporting MXF data to other systems.

When an application needs to convert the contents of an MXF file to and from other formats, such as AVI, the entire file will normally need to be unwrapped and re-coded in the new format. Often the essence itself (for example, MPEG long GOP video) will not need re-MPEG encoding; however, it is very likely that metadata will be lost when an MXF file is converted to another format.

3.8 What is meant by simplicity?

MXF files must be amenable to implementation in high throughput hardware or software devices. This translates into the need for well-defined design parameters for buffer size, latency, and the need for algorithmic simplicity. MXF is also intended to cover a very large application space, and not all the requirements apply to all the applications. The examples below are all application specific:

Example constraints:

- Buffer size must be minimized for low latency streamability.
- KLV wrapping and file partitioning latency must be small and bounded.
- Algorithms should not require distant look-ahead to calculate parameter values.
- Algorithms should not require deep stacks or high performance coprocessors, and should preferably be straight-line (no looping).
- Operational patterns should create controlled and bounded application environments that are constrained enough to ensure interoperability, yet broad enough to allow many implementations.

The design can also be kept simple through the proper use of layering. Network, transport and session layer functions and data units must be kept separate at all costs, so as not to burden any layer with processing that belongs to another layer.

3.9 Why does MXF need to work with stream interfaces?

MXF files will often be processed in streaming environments. This will include streaming to and from videotape and data tape, and transmission over unidirectional links or links with a narrow-band return-channel.

In these environments it is impractical to rewind the stream to update parameter values so files must be written sequentially. This implies that the minimum buffer size and latency are determined by (among other things) the maximum KLV packet size. Implementations of MXF streaming should take into account all the constraints of the operational pattern in use, as well as extra restrictions imposed by the particular streaming data link before recommending buffer sizes or latency requirements.

Sequential writing is necessary when source or link or destination operate only in streaming mode. Random access writing is permissible before or after data transfer, for example, to optimize downstream access performance.

Operational patterns have a special qualifier bits that indicates that the file has been created for streaming.

3.10 How does MXF provide for stream recovery?

Streaming environments also impose requirements for recovery and re-synchronization in several different circumstances:

1. When a packet or other data block is lost.
2. When a decoder joins a transfer that is already in progress.
3. When a transfer or partial transfer is restarted.
4. When it is necessary to access or retransmit a file that is still being received ("pre-play").
5. When overall metadata is modified during the time of transfer.

The first of these (packet loss) usually requires a return-channel or forward error correction for effective protection. The other circumstances are addressed by judicious design of the format to allow for re-synchronization points and for repetition of important metadata.

3.11 How does MXF provide for application diversity?

Different applications may require metadata to be processed separately from the essence. Other applications (such as archive) may require metadata to be stored with the essence. This requires efficient insertion and extraction of the metadata from the essence container(s) of the file.

Some applications may prefer index tables to be accessed separately from the essence; others may require the two to be accessed together. In some cases, the index tables are most naturally stored at the start of the file; however, while recording, the most natural location is at the end of the file. This diversity requires efficient insertion, extraction and relocation of index tables within the file.

3.12 How does MXF make references to its different components?

MXF uses different referencing mechanisms for different purposes. One example that causes confusion is the difference between references to the top-level "file package" and "the essence". The MXF content storage set uses Instance UIDs to reference all the packages in an MXF file. One of these will match the Instance UID of a file package within the file. This is a strong reference to the package. The package itself is a description of the essence, but is not the essence itself.

The content storage set also uses Instance UIDs to keep a list of essence container sets. These are used to group the various IDs that enable an MXF decoder to work out which partitions and index tables relate to which top-level file package. Specific details are given in section 7.5.

This seems straightforward until we look at how a material package SourceClip references the essence. This structure does not use the instance UID values, it uses the 32-byte UMID of the essence as a reference. This is because the material package is referencing the essence of which the top-level file package is a description.

4 File interchange requirements

There are two basic types of file interchange requirement: User requirements and technical requirements. The user requirements are lists of things that users want to be able to do with files. The technical requirements are features of the file that allow applications to be accommodated.

4.1 User requirements for an interchange file

The MXF format, at its lowest level, should support functionality that is commonly available in today's video file-servers. The MXF/AAF Joint File Interchange Working Group in co-operation with the EBU P/PITV group and the SMPTE have summarized the user requirements for MXF as follows:

Table 1 – User requirements table

User Requirements	General Priority LIST	PROFESSIONAL APPLICATIONS			
That are assigned the following priorities: A = Baseline ("Must"), B = Enhanced ("Can"), C = Extended ("May"), U = Undecided or not determined, X = not allowed (should not be allowed)		**Publication** (Emission, Transmission, Store & Forward, etc.)	Content Repository	Finished Interchange	Authoring Interchange
Must be easy to understand & apply and standardized	A++	Y	Y	Y	Not easy
Must be compression independent	A	Y	Y	Y	Y
Low implementation overhead	A	Y	E.g. Could be complex if editing required	Y	No
Must be open (as per ITU definition)	A	Y	Y	Y	Y
Must provide Identification of the payload	A	Y	Y	Y	Y
Must provide for normative templates	A	Y	Y	Y	Y
Must be extensible in header and body (by KLV coding?) (E.g. from one frame to many frames)	A	Y	Y	Y	Y
Scalability (small file/single frame to large file)	A	Y	Y	Y	Y
Must provide synchronization for multiple essence types e.g. Audio/Video/Data Essence and certain Metadata	A	Y	Y	Y	Y
Must wrap Video Essence[s] Audio Essence[s] Data Essence[s] Metadata	A	Y	Y	Y	Y
Must permit direct mapping for existing transfer format (e.g. MPEG-TS, SMPTE 314M, FC-AV, ATM-Wrapper)	A?	Y	Y?	Y	Not always needed
Must uniquely identify container framework (e.g. FC/AV)	A	Y	Y	Y	Y
Must be usable on major platforms / OSs	A	Y	Y	Y	Y
Must be application independent	A	Y	Y	Y	Y
Must provide means for partial file transfers	A	Y	Y	Y	Not always needed
Must provide means for graceful recovery after interrupted transfer	A	Y	Y	Y	Desirable
Must provide cut-only edit capability (versioning)	A	Y	Y	Desirable	Desirable
Must be transport and storage mechanism independent (e.g. FEC is a transport issue)	A	Y	Y	Y	Y
Simple and complex template (backward-forward compatibility?)	A	Y simple	Y Both	Y simple	Y complex

User Requirements	General Priority LIST	PROFESSIONAL APPLICATIONS			
That are assigned the following priorities: A = Baseline ("Must"), B = Enhanced ("Can"), C = Extended ("May"), U = Undecided or not determined, X = not allowed (should not be allowed)		**Publication** (Emission, Transmission, Store & Forward, etc.)	Content Repository	Finished Interchange	Authoring Interchange
Format Expandability in Operational Patterns: 1a: Simple Pattern: single item/representation (e.g. clip) Extended Pattern that might be an individual pattern or a more generalized pattern 1a. A: Compiled: Segmented item/representation (e.g. part of a final composition) B: Uncompiled Program: simple edit representation (e.g. compound clips=each track has its own time line) C: Uncompiled Compound: edit representation (as template before but with handles e.g. for cross fades) D: Uncompiled Elements: E: Metadata only representation F: Effect representation G: Archiving Etc. Prerequisite for all Operational Patterns, generalized patters etc. is a proper standardization/documentation to guarantee interoperability. It is also assumed that Operational Patterns reflect a certain application(s) environment. This has to be described in the documentation (standards).	A	A, B, C,	A, B, C, D, E	A, B, C	A-F
Easy conversion from file to stream and vice versa	A	Y	Y	Y	Desirable
Robustness against errors. Examples: During file transfer interrupt; Corrupted header File access error;	A	Y	Y	Y	Y
Interface to pre-existing interconnect standards (mappings into IP, FC etc.) Note: robustness against errors may belong more to the transfer mechanism than to the file format domain.	A	Y	Y	Y	Y
Extensibility to include non-predefined data (e.g. dark Metadata)	A	Undesirable	Undesirable	Undesirable	Y
A/B LIST					
Can provide random access: Play/access while transfer Play/access while record (open ended)	A/B	Y	Y	Y	Y
Fast frame and field level access (E.g. by means of indexing to field/frame/audio frame level)	A/B	Y	Y	Y	Y
B-LIST					
Low latency (values see 1st TF report) "goal 1 Frame, 1 GOP"	B	Y but depends on further applica-tions	Maybe	Y	Maybe
Link Metadata to structural composition information	B	N	Y	Maybe	Y
Can accommodate a range of GoPs (e.g. MPEG)	B	Y	Y	Y	Y

User Requirements	General Priority LIST	PROFESSIONAL APPLICATIONS			
That are assigned the following priorities: A = Baseline ("Must"), B = Enhanced ("Can"), C = Extended ("May"), U = Undecided or not determined, X = not allowed (should not be allowed)		**Publication** (Emission, Transmission, Store & Forward, etc.)	Content Repository	Finished Interchange	Authoring Interchange
Can provide for re-coding data sets (e.g. compression history information)	B	Y	Y	Y	Y
Can provide Index i.e. can tabulate byte offsets within a file that correspond to given Time codes.	B	Optional	Optional	Optional	Y
Assignable granularity of Metadata (field, frame/clip/file)	B	Y	Y	Y	Y
C-List					
Extensible for internet. Metadata as binary and text format	C	Y	Y	Y	Y
Discontinuous essence elements (chunking)	C	If required Y	If required Y	If required Y	If required Y
Allow externally referenced essence files for certain applications such as Archiving. A proper standardization / documentation is prerequisite if external references are used.	C	N	Undesirable	N	Y
X/U List					
Allow proprietary vendor created templates	X	?	?	?	Maybe

4.2 Technical requirements of a file

The technical requirements derive from the user requirements. The individual requirements are introduced gradually throughout the document. A typical example of a technical requirement is that of body partitions. The user requirements state that the file format must support partial transfers and must provide graceful recovery after errors. The technical requirement from this is that the file must periodically contain repeated data to allow partial transfers or recovery. The implementation chosen in MXF is body partitions.

5 A guide to the wording of the MXF standard

MXF files apply a subset of the AAF class model. Because of this, many of the words used in the MXF standard are the same as in the AAF specification. There are occasionally subtle differences of meaning between MXF and AAF because of the different applications they address. An example of naming differences is the use of the package naming where in AAF the phrase “file source mob” is used, the shorter MXF phrase “file package” has the same meaning. In all MXF documents, new normative terms will be defined within the document. Subtle differences with AAF have not been highlighted in the specification because the MXF standard is self-consistent. The main glossary of terms and data types can be found at the start of SMPTE 377M.

5.1 Normative vs. informative

5.1.1 Normative

The definition of normative is given in the SMPTE Administrative Practices. For information, normative parts of a document cover those elements of the format that are fully specified. The implication of a normative clause is “if you do this particular function or encoding process, do it like this”. Normative does not imply that

all decoders must understand all normative elements, just as it does not imply that all encoders will encode all normative elements. Normative clauses use the verb “shall”.

The value of a normative clause is that it defines the parameters and syntax for a given function or process.

5.1.2 Informative

Informative parts of a document provide additional explanation or describe optional functions or processes. The implication of an Informative clause is “you may do this particular function like this”. The value of an Informative clause is that it provides an illuminating example of how to achieve a function or process to improve interoperability. Informative clauses use the verb “may”.

Since neither normative nor informative convey any information as to which functions an implementation is expected to perform, additional terminology is needed.

5.1.3 Recommendations

There are many recommendations in SMPTE 377M. There are many places where it was desirable to make a normative provision, but the provision could not be enforced. For example “the duration property should be correct in all header metadata repetitions”. Devices such as cameras cannot create an MXF file with the correct duration because the header is written before the file is closed and completed. This provision is, therefore, a recommendation rather than a normative requirement. Recommendations use the verb “should”.

5.2 Encoding and decoding

One of the key points of developing any new techniques is to consider the layering of any file format and its contents. This helps us to understand the meaning of an ‘encoder’ and a ‘decoder’ at any given layer. Unfortunately attempts to introduce new words such as “encapsulate” have not been well accepted and words such as “encoder” are forced to have slightly different meanings depending on context.

The layers for encoders and decoders can be broken down as follows:

Table 2 – Content layering

Layer	File Body	File Header & Footer
Application	Source Coding (525, 625 etc)	Data Interpretation (e.g. dictionary of data definitions)
Essence Coding	Compression Coding (MPEG, DV etc)	Data Communication (e.g. relationships between objects)
Container	Essence Container (CP, PS, TS for MPEG, DIF for DV)	Data Container (e.g. KLV sets, Objects)
Encapsulation	MXF coding (KLV)	
Transport	Transport (IP packets, etc)	

The overall system is as follows:

1. An MXF system *accepts* essence represented in its "source coded format".
2. The essence is optionally *compressed* through a source **encoder**.
3. The essence components are *multiplexed* by an MXF **encoder** into partitions.
4. The multiplexed partitions are *encoded* into an MXF file by an MXF **encoder**.
5. The MXF file is *decoded* by an MXF **decoder** to present the essence to a user.
6. The MXF file is *demultiplexed* by an MXF **decoder** to split the file into its different essence components.
7. The encoded essence is *decompressed* by an essence **decoder**.

8. The decompressed essence is *displayed* or *presented* in its "source coded format".

NOTE – Processes are *italicized*, nouns are in **bold**.

Note that not all processes will be supported by all equipment. Many devices will operate over all layers to provide a network or stream interface at the lowest layer, and an interface to the user at the highest layer. However, devices that simply 'store and forward' need only respond to the lowest two layers and devices that 'unwrap' the data contents to provide the raw data streams only respond to the lowest three layers.

5.3 Functional descriptions — Encoder required, etc.

The following terms have been proposed to describe functionality that must be supported in order to create an interoperable MXF environment. SMPTE 377M defines the normative terms, extra text and words are given here for information.

Summary:

Table 3 – Functional descriptions

Phrase	Abbreviation	MXF encoder	MXF decoder	Meaning
Required	Req	Shall	Must	See below
Encoder Required	E/req	Shall	May	See below
Decoder Required	D/req	May	Shall	See below
Optional	Opt	May	May	See below
Best Effort	B.Effort	Should	May	See below
Dark	Dark	Should not	Shall ignore	Used to describe essence and metadata items that are unknown to an application at a given time.
Incompatible	Incompat.	Shall not	Can explode	Items that could cause catastrophic decoder failure

5.3.1 Required

A **required** item is essential to both encoder and decoder. An example of a required metadata item is a preface set. The encoder must encode this and the decoder must understand it and act on it.

5.3.2 Encoder required

An **encoder required** Item must be sent by the encoder, but a decoder may choose to ignore it. An encoder required item must be encoded by the encoder, but need not be decoded by the decoder. An encoder must not assume that a decoder has taken notice of such an item.

5.3.3 Decoder required

A **decoder required** item may be sent by the encoder. If sent, the decoder must act upon the Item. If not sent, then the decoder **may** either do nothing, or set the item to an **default** value or take a predefined default action if specified by the relevant document.

5.3.4 Optional

An **optional** item may be sent by the encoder if it is known. If sent, the decoder may choose to ignore the Item. If not sent, then the decoder **may** either do nothing, or set the item to a **default** value or take a predefined default action if specified by the relevant document.

5.3.5 Best effort

A **best effort** Item is very important to a decoder, but may not be known by the encoder at the time of file creation. These Item have distinguished values that mark them as not known; when these distinguished values are used, the file becomes an "incomplete" file as explained in section 3.5.2.

Note that a 'default' value for an item is the value that a decoder should use in the absence of the Item. A 'distinguished' value is used by an encoder to signal that the Item value is unknown by the encoder. The difference between 'default' and 'distinguished' is important.

5.3.6 Dark

A **dark** Item is one that is unknown by a decoder or an encoder. This Item may be proprietary and unknowable by a decoder. It may be an extension to SMPTE 377M that has not been incorporated into a device or application. It may even be metadata in the original specification that is not relevant to a device or application. All that is certain is that the meaning of the metadata is unknown. In certain application environments, encoders may be required to carry dark metadata and decoder may be required to make dark metadata available.

SMPTE 377M uses KLV local sets with 2 byte tags and 2 byte lengths and includes a special pack structure called the "primer pack" to ensure that dark metadata properties can be created and handled without the possibility of a numerical clash of local tag values.

Why is this important? Imagine that two companies, X and Y, each independently, want to extend the MXF identification set to include some vital property of their application in every MXF file that they save. Without the primer pack, there is a finite chance that they will both choose the same local tag value for their private metadata property and when they open each other's files, they will misinterpret or even corrupt each others' metadata properties. The primer pack mechanism exists to prevent this happening.

5.3.7 Incompatible

An encoder must not send incompatible items. This data classification is provided to allow certain data items to be forbidden if they could prevent successful or deterministic decoding. There are no "incompatible" items defined within SMPTE 377M, but the concept of Incompatible Items is described here because it gives a common word for designers and implementers to describe a class of metadata that should be avoided.

5.4 Element, item, container, stream, body, multiplexing and interleaving

An MXF file may have external essence in addition to essence within the MXF file body. The MXF file body may have several essence containers that are multiplexed together, each of which can sometimes be called a stream. Each of these essence containers may have a single piece of essence or may have different essence elements interleaved together. Each of these elements may be categorized into picture items, audio items, data items and system items. This results in an MXF file body that may contain a multiplex of essence containers that in turn contain interleaved essence items that in turn contain the individual interleaved essence elements.

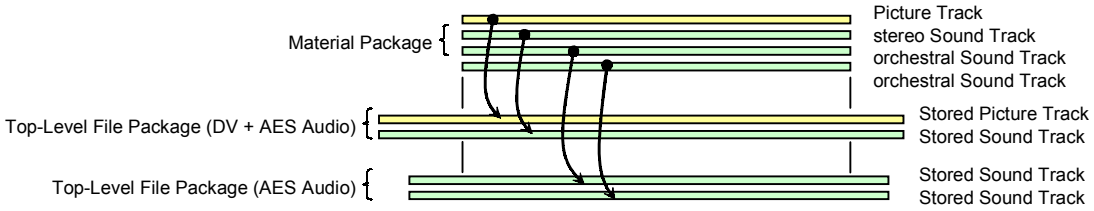


Figure 4 – Multiplexing and interleaving — Logical view

That was horribly complicated, so an example will help to clarify this extreme example of MXF capabilities. Imagine a file that was captured in DV and has had its stereo sound extracted and separately edited. Later in the process, an orchestral score was added using in a separate essence container and described by a separate file package. The operational pattern 1b mechanism is used to synchronize the two file packages. The resulting file looks logically like figure 4. This seems a simple logical view, but the physical representation is much more complex as shown in figure 5.

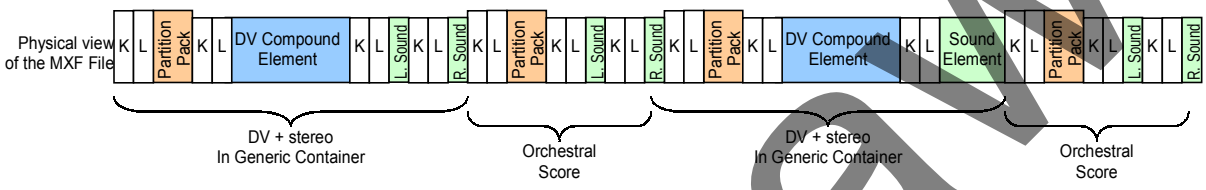


Figure 5 – Multiplexing and interleaving — Physical view

The final sentence of the opening paragraph may now be a little clearer. The file is a multiplex of different partitions; in this case two generic containers are multiplexed using the partition mechanism. One of these generic containers is an interleave of essence items — a DV compound item and a sound Item. In each of the multiplexed generic containers, the sound items contain an interleave of sound elements — left and right channel. It is also worth noting that the DV itself is an intrinsic interleave of DV-DIF blocks. In most MXF processes, this level of interleaving is left to the essence codec and is usually opaque to MXF.

There are normative descriptions of these words in the format document and in the generic container document. It is strongly recommended that new essence container documents follow this wording.

5.4.1 Essence element

In many places in SMPTE 377M documents, the term essence element is used generically.

In many discussions of low level wrapping of the data in a generic container mapping, the term essence element is used to mean “a KLV wrapped essence entity that has a defined key”. For any given key, any essence elements with that key relate to the same essence stream.

In other macroscopic discussion of interleaving and multiplexing, the term essence element is used to describe all the KLV wrapped essence entities with a given key, such as a single video data stream. When a stream has a single video data stream and an associated audio data stream, the essence container would be regarded as having two essence elements, regardless of how many KLVs were used to hold the essence.

This contextual use of the term essence element may cause confusion, but the authors felt it would be worse to try to invent a new term for every one of the subtle changes in context.

5.5 Classes, objects, packages and references

In this section, the concept of object implementation will be introduced, as will the idea of collecting objects and information into packages. This section is intended to **improve understanding of the concepts**. It is not

intended to be a rigorous definition of the terms. The actual definitions of packages, strong references and the like can be found in SMPTE 377M and other MXF documents.

5.5.1 What are classes?

A class is a generic definition of the behavior and properties of a generic object. The textbook example is a given make and model of a car. All the cars from the same class have the same generic behavior and properties. When describing a particular car, all the properties (such as color, engine size) are given values. This is called an object or an instance of the class. The class definition includes all the core design parameters that are common to all instances. A given make and model of a car (i.e. the class) may be a blue or red but are still clearly the same car, except the color 'property' has been changed between the two objects.

Classes are defined by a set of data items, where each item is commonly called a property. When an instance is made from a class, it becomes an object and values are assigned to all the properties.

Modeling of a system can involve the creation of many similar classes. In this document, we have described that there are different sorts of track. Each of these tracks has properties that are very similar. In modeling terms, there is an abstract super class that defines the common functionality of all the different tracks. Abstract means that the class is never used directly. Super class means that the purpose of this class is to create subclasses that add to all the properties of the super class. A generic track is an abstract super class. A timeline track and an event track are two subclasses that share all the common properties of the track class and have added their own specific properties and behaviors.

5.5.2 How are objects implemented?

In MXF objects are implemented as KLV local sets as defined in SMPTE 336M. In SMPTE 377M, the word set is used in nearly all cases to describe an object of a given class. The specification of the class properties is done using tables in the normative MXF documents, and the behavior is specified in the text of the document.

5.5.3 What is a package?

A package is simply a container for a number of tracks that in turn represent the passage of time. The package mechanism allows different tracks to be "ganged" together in parallel. This allows metadata and essence to be synchronized to a common timeline.

Each package describes some aspect of the essence or data in a file and the different types of package will be explained here with the help of some real world analogies.. The top-level file package contains a collection of metadata items and sets that describe; for example, the embedded video essence. It is described as though the essence tracks were in a file — hence the name top-level file package.

It is important to note that the tracks are synchronized in time. This synchronization is determined by a specified Offset value from the beginning of each track.

For the AAF-conversant reader, it is useful to note that composition packages are not currently used in MXF.

5.5.3.1 What is the material package?

The material package is a metadata structure that generally represents the output timeline of the file. If you imagine the file being "played" in an MXF player, you would expect to see video, hear audio and view the data as though it were a tape in a VTR. The material package contains the "hooks" that allow this to happen. It contains timing information about the output — for example, how the time is measured. It contains information about the output tracks — how many and what format they take. It also provides hooks to say where the essence data comes from to fill these tracks (i.e., which top-level file packages).

As can be seen in figure 6, the material package can be viewed as a set of parallel tracks — one for each kind of essence in the output stream. There is metadata associated with the file that has a global scope, such

as the name, the UMID, etc. Each track contains further metadata to describe the way in which the final output should be created from the top-level file packages.

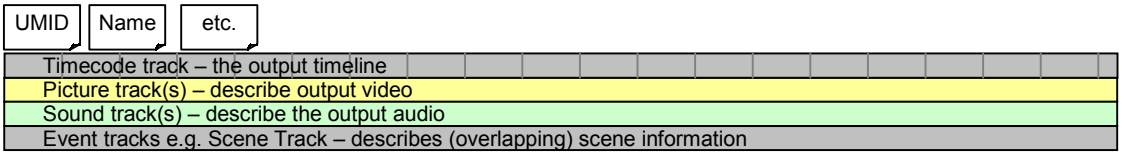


Figure 6 – Material package

Figure 7 shows the relationship between the pictures. It shows how the material package track can define a sequence of SourceClips. Each SourceClip in the material package indicates which portion of a top-level file package should be "played" next. This is the way in which MXF supports edit decision lists (EDLs).

The material package in figure 7 shows how the SourceClip references the entire top-level file package. Only the file packages in the top level of an MXF file describe the actual essence in the file body.

The MXF operational patterns constrain the relationships between the material package SourceClips and the file package(s) in an MXF file. In an OP1a file, there is no EDL support and the material package references the entire top-level file package. In an OP3c file, complex timeline relationships are allowed that may require the MXF decoder to have random access capabilities.

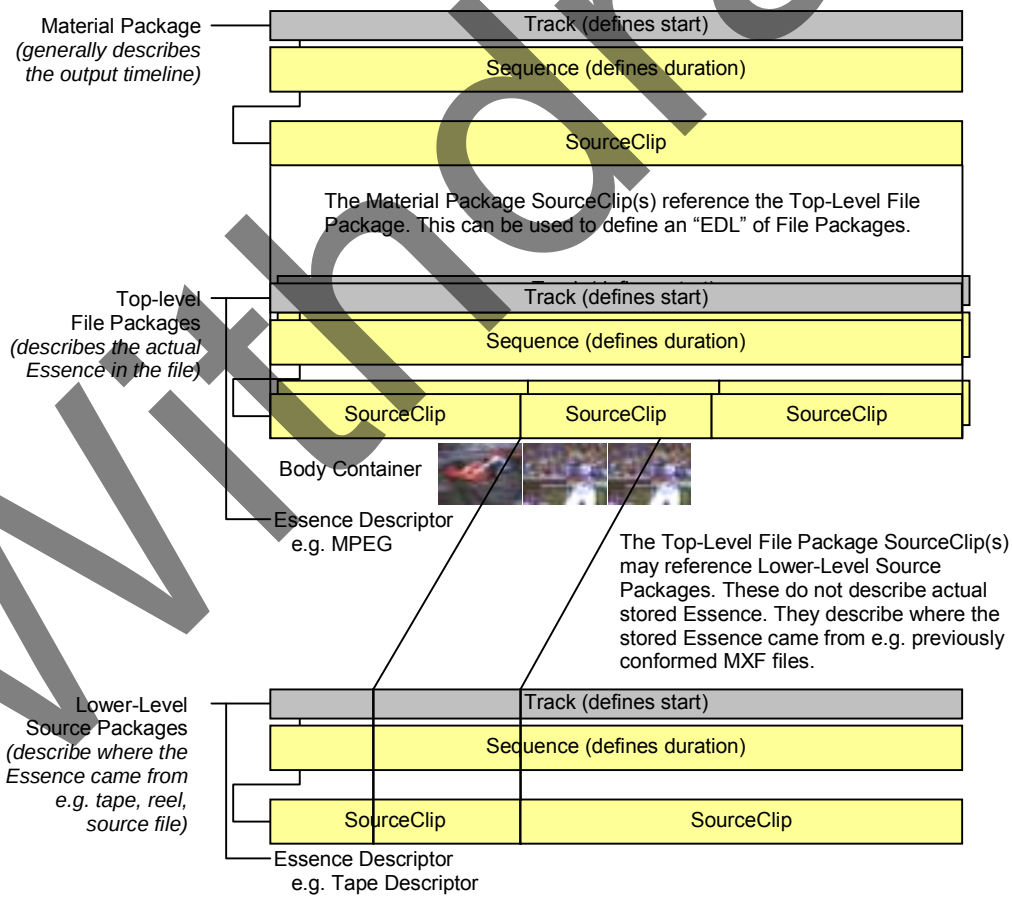


Figure 7 – Relationship between the packages

5.5.3.2 What is a top-level file package?

The top-level file package represents the storage of some essence. This essence may be stored in the file body or externally in a separate file (located by information in the essence descriptor). The top-level file package contains the tracks that describe the type of essence, the compression scheme used (if any) and the source coding parameters such as the number of samples, pixels and aspect ratio of the essence as appropriate.

The tracks in the top-level file package may be made up from a number of SourceClips that are used as historical annotation to indicate where the content came from.

5.5.3.3 What are lower-level source packages?

The SourceClips in the top-level file package may refer to either file packages or physical packages. In SMPTE 377M, the generic class “source package” is used to refer to either File packages or physical packages.

In SMPTE 377M, a source package that is not at the top level is used to describe the derivation of the essence; i.e., where it came from. This is very useful metadata, especially when creating archives or providing historical information about the source of the file package. Lower-level source packages often contain physical descriptors such as tape descriptors that refer to a physical location or storage medium for the content.

5.5.4 References

Within the MXF format we need a way of referring to objects. For example the statement, “A material package has one time code track object”, is quite clear. This is known as a strong (one to one) reference between the material package and the time code track object.

Each metadata set is coded and identified as a KLV local set and has a value that contains all the locally coded metadata items in sequence as a tag (typically 2 bytes), length (typically also 2 bytes) and the individual metadata item value. Note that most MXF sets contain a unique identifier (instance UID) for that set. This Instance UID is the core data construct used to connect objects together into a logical framework

A “strong reference” to any KLV coded data set is a one-to-one relationship between the reference and the target data set. In MXF files, a strong reference is made by matching the value of a “StrongRef” in the referencing set to the Instance UID property of the referenced set.

A “weak reference” also uses an instance UID to connect data sets, but any weakly referenced data set or item may be referenced by more than one other data set. Thus, a weakly referenced set is a stand-alone data set with an Instance UID to which one or more other data sets can refer through the value of a ‘WeakRef” property. In order to properly construct an MXF file, each and every set must have one strong reference to it. There is no limit to the number of weak reference which may be made to a set. Figure 8 illustrates the concept of strong and weak references in a stream of KLV coded metadata sets.

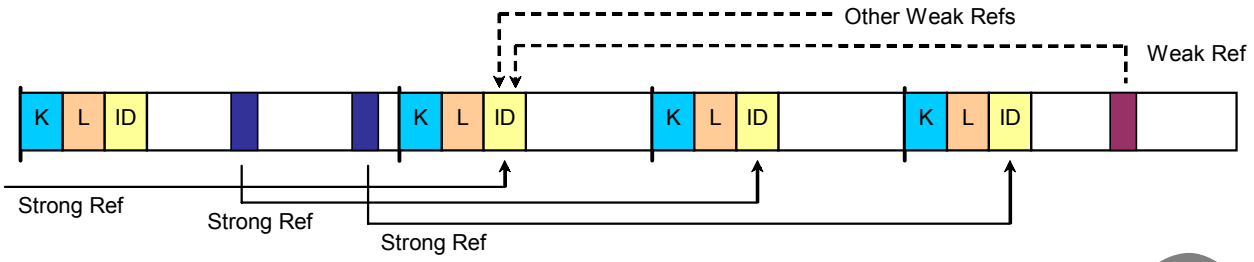


Figure 8 – Strong and weak referenced data sets in a KLV coded data stream

Note that the metadata sets are contiguous in order to preserve the KLV coding protocol (i.e., there are no gaps between the metadata sets.

Figure 9 provides a more detailed example of data set organization and includes three techniques for the connection of data sets:

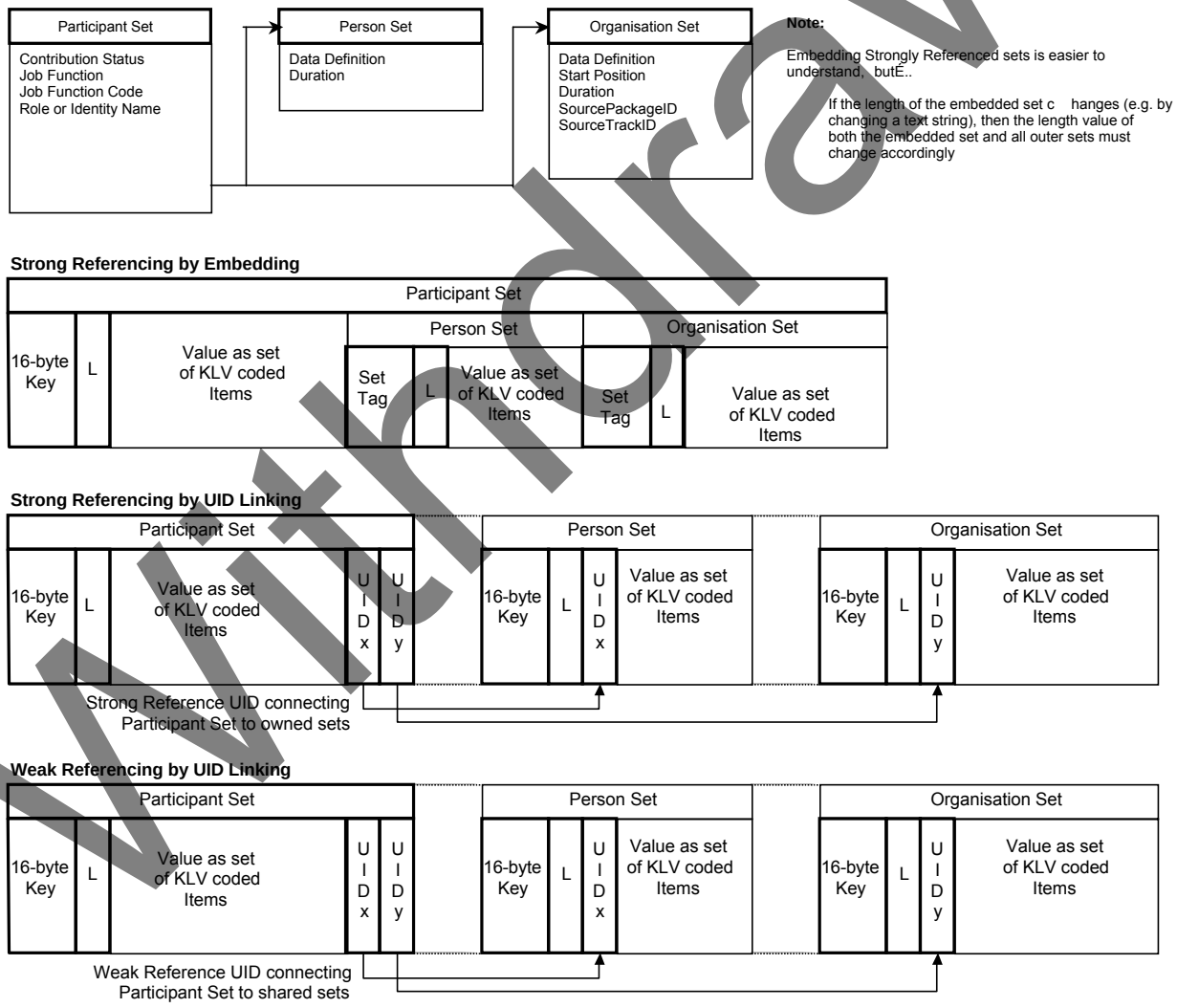


Figure 9 – Strong and weak reference data sets in streams

Strong referencing. Strong referencing implies ownership of the referenced object as well as a one to one relationship with it. When an MXF application creates a tree of interlinked objects starting at the MXF preface set, all objects will have at least 1 strong reference so that they are “owned” and can fit into the overall tree. An object may additionally be weakly referenced by a large number of other objects.

Strong referencing by embedding. This can be used where a strongly referenced data set is easily embedded into the referencing data set. It is used in applications requiring high-speed operation, but has the drawback that when the referencing set is changed, the length fields of both the contained and surrounding sets must change accordingly. This mechanism is not used in the MXF header metadata, but may be used in an essence container specification. Ownership of the referenced object is implicit because it is contained within the referencing object.

Strong referencing by UID. This requires an Instance UID property in the referenced data set and a property of type StrongRef in the referring data set. The overhead is thus higher than the embedding method above, but if a property value in the referenced set changes length, it impacts only that data set and its parent data sets, but does not affect the length of the referencing data set.

Weak referencing by UID. Weak referencing uses an Instance UID in the referenced data set; one or more other data sets can refer to the referenced data set by using the same Weak reference UID value. The advantage of a weak reference is that the values of metadata items in a data set can be shared by several referring data sets. It is worth noting that everything within an MXF file that is weak referenced must also be strongly referenced.

A **reference collection** is a list of UIDs connecting the referencing entity to zero or more other entities (either weak or strong).

A **reference array** is a set of ordered references (or vector). This implies that the order is significant for whatever reason.

NOTE – Because all properties in MXF are unique within the AAF class model, all StrongRef and WeakRef properties are strongly typed. This means that the property can only have a StrongRef to a specific sort of Set (or one of its subclasses). Thus, SMPTE 377M uses the nomenclature “StrongRef (MyClass)” to mean a strong reference only to an object of type “MyClass” or an object derived from MyClass.

For every reference in an MXF file, an MXF decoder should be able to find a set that is the target of that reference. The previous sentence uses the word “should” and not “shall” — why? From the definitions above, you would expect that a decoder would always be able to find the target of a strong reference. In the absence of any extensions to SMPTE 377M, this would be a true statement; however, it is expected that additions will be made and new metadata sets and schemes will be developed as the format matures. Decoders that do not understand these extensions are likely to discover that there are dark metadata sets (i.e., the set key of the KLV is not understood by the decoder) within the file and that there are references without identifiable targets.

“Clever” decoders may be able to help in this situation, by looking inside dark sets, especially those whose local tags appear to be stored in the primer pack. Instance UIDs could then be discovered with some high confidence and the presence of dark extensions to SMPTE 377M discovered. In some circumstances, this behavior may be quite helpful, but in general, making intelligent guesses about dark sets is outside the scope of SMPTE 377M. It may also lead to unpredictable results.

To summarize the MXF referencing behavior:

1. References are made from a property in one set of type WeakRef or StrongRef to the InstanceUID property in another set.
2. All header metadata sets (other than the primer) are linked to the preface (directly or indirectly) by strong references.
3. All strong references in any instance of header metadata match one and only one set in that instance.
4. Weak references may be made to "global definitions" that are outside the file, in these cases the WeakRef will be either a UUID or a UL. Therefore, if a weak reference cannot be matched in the file it can be regarded as a global definition.

5. Typical global definitions are codec ULs, container ULs and compression ULs, which are used to enumerate different codec, container and compression mechanisms
6. As dark metadata can exist in the header this means that references of any kind may appear to be unresolved even though they are correct. MXF decoders must be able to cope with this.

5.5.5 Resolving ULs and UUIDs

SMPTE 377M contains a large number of sets and properties (referred to as items). The normative definition of these properties — what they are and their type (e.g., integer, UL, string, etc.) — is given by the SMPTE metadata dictionary (RP 210). In the MXF format document, bytes 9 onwards of the entry in the dictionary are repeated in the "UL designator" column of the set definitions. Within the file, local set coding is being used in which a short 2-byte tag is used to substitute for a 16-byte UL.

Some of the properties in SMPTE 377M are themselves Universal labels. Some of the values that these properties may take are ULs, and indeed some of the KLV keys in MXF may be ULs. These labels are generally used to identify lists of unique things. For example "picture coding type" has a UL value. All the picture coding types that are known to MXF are simply listed in the SMPTE labels registry. Applications that need to determine the meaning of a label use the SMPTE labels registry as the normative reference.

In certain cases an encoder may place an un-registered UL or a non-UL unique identifier in a property of type “UL”. Example cases are where new MXF features are being developed, but have not yet been standardized, and where private extensions are added for use in a carefully controlled MXF system. Some of these cases are outside the scope of the MXF format, but decoders should make every effort to handle these files gracefully. For example, decoders should not rely on the values being validly coded as a registered SMPTE label.

5.5.6 UUID properties and their scope

Universally unique IDs (UUIDs) are arithmetically computed unique numbers that can be used in MXF files in two different ways. Firstly, they are used for making links between different parts of the same file, such as with strong and weak reference Instance UIDs. Secondly, they are used to provide identifying or typing information, such as where a property’s local tag is translated via the primer pack into a UUID. In the first case, the UUIDs have partition scope; an occurrence of the same UUID in two different files, or even two different partitions of the same file, does not imply any relationship between them — even thought the likelihood of the same UUID being generated twice is extremely remote. In the second case the UUIDs have global scope, wherever the same UUID is used it has the same meaning.

5.5.7 Byte order of UUIDs

ISO/IEC 11578 states that in the absence of explicit specification to the contrary, UUIDs are encoded as a sequence of 16 bytes starting with the bytes holding the time field and ending with the node ID. However, the significance of byte order depends on the scope of the UUID.

If the scope of a UUID is local to the file then the byte order is unimportant, providing each occurrence of that UUID uses the same byte order. In these cases the default order specified in ISO/IEC 11578 should be used.

Where UUIDs have global scope the byte order is significant. In these cases the byte order will be given when the UUID value is published. For example, where a manufacturer publishes the UUID that a particular device inserts into the “product UID” field in the identification set, the byte order of that UUID will be specified as well as the values of the bytes.

5.5.8 Storing ULs and UUIDs in the same property

Some data fields, such as the UID property of the LocalTagEntry batch in the primer, can contain either a UL or a UUID. In this case there is an advantage to using a particular byte order for the UUIDs. All UUIDs have a 1 in the most significant bit of the “clk_seq_hi_res” word (byte 9), whereas all ULs have a 0 in the most

significant bit of the first byte. If UUIDs are stored with a byte order that places the “clk_seq_hi_res” word first, then it is always possible to tell if the value is a UL or a UUID by examining the MSB of the first byte. This byte order also prevents the remote possibility of a UUID being stored that matches a registered UL. For these reasons, it is recommended that when any UUID is published for inclusion in a data field that can also contain ULs, the byte order specified for that UUID be the same as the ISO/IEC 11578 default order, but with the upper and lower eight bytes swapped.

The section above gives rise to the following guidelines:

- A UUID may be stored in a data field of type UL by swapping the top and bottom 8 bytes of the UUID (the most significant bit of the first byte of such a swapped UUID is always 1);
- MXF decoders should accept a swapped UUID in a place where a UL is expected.

NOTE – AAF uses a compatible byte-swap method for storing ULs and UUIDs in the same properties, which it defines as AUIDs.

5.5.9 ULs identifying the file’s handling requirements

In the partition packs of the MXF file, there are a number of properties whose UL values are intended to give an indication of the codec and handling requirements needed for the file. This information is intended to be a performance enhancement to provide “fail-fast” functionality. This information is located in the first few bytes of every file so that an application can quickly determine if it is able to handle the content of a file. The information is copied from the authoritative information in the header metadata.

The operational pattern UL identifies the timeline complexity of the file. The essence container ULs identify the essence data that is contained in the file so that an application can determine if a suitable codec is available. These numbers are registered values so that an application that cannot handle a particular essence container type is able to report the essence type in the file. This type of reporting behavior helps users to identify content and is encouraged. Anonymous failure such as “a codec cannot be found” without reporting what sort of codec was sought is not encouraged. Older decoders that are unaware of new UL values should at least attempt to report the ULs that were not known. It is important to note that it may not be possible for this information to be provided by all MXF encoders and that decoders should not fail if this information is empty or missing.

If an MXF File contains multiple essence containers, but these are all of the same type, then the essence container label appears in the partition pack only once. This non-duplication is to ensure that a higher operational pattern file with 100 small MPEG clips need not insert 100 ULs in the list.

Some essence container specifications (such as the MPEG long GOP generic container mapping) define essence container ULs for the different MPEG streams that may be encountered when transwrapping from MPEG program stream to MXF. It is possible that the list of essence containers will contain a UL for the sound data and a UL for the picture data even when the resulting file contains only a single essence container with interleaved sound and pictures. During the design of MXF it was felt that there needed to be a descriptor for each of the different types of audio so that the MXF decoder requirements could be determined rapidly.

As an example, if you have an OP1a MXF file with MPEG-2 video, two channels of AES audio and time code, the file would have:

- 2 ULs in the EssenceContainer list (1 video, 1 audio);
- OP1a declared in the partition pack and the preface set;
- 4 tracks: 1 picture, 2 sound, 1 time code;
- Material package tracks have the same duration as the top-level file package tracks.

MXF decoders must be able to cope with the case where there are many essence containers of the same type with a single UL in the EssenceContainer list. MXF decoders must also be able to cope with the case where there are several ULs in the EssenceContainer list, each of which relates to a different element of a single MXF generic container.

5.5.10 Data definitions

There are several MXF sets that are generic (e.g., the sequence set) and the specific behavior is identified by the data definition property. In AAF, these components are implemented as weak references to definition objects in the dictionary. These definition objects each contain an Identification property that is a 16-byte "magic number" that the application can use to figure out how to handle the component.

MXF does not have such a dictionary, so it cannot work the same way. Instead the DataDef property in a component actually **is** the 16-byte "magic number" that the application can use to figure out how to handle the component. This is a very subtle change in behavior between AAF and MXF, and implementers of compatible systems should take appropriate actions to ensure interoperability. This type of data is actually a weak reference into an external data set — i.e., a registry or dictionary, such as the SMPTE labels registry.

5.6 Implementing objects as sets

KLV coding allows related metadata items to be grouped together in sets; e.g. Titling metadata might be grouped into a set for convenience. SMPTE 336M defines several mechanisms for grouping the data together. Basically, a set comprises an outer KLV that defines the set and a number of inner KLVs that define the data items.

The inner keys could be full length (Universal set) or could be shortened for processing and storage convenience. KLV sets using these shortened item keys are known as local sets and the technique is fully defined in SMPTE 336M. This standard defines how all sets have Universal labels with a consistent definition in the first 8 bytes of the type of data set or data pack being used. The options provided are:

- Universal set;
- Local set;
- Variable length pack;
- Fixed length pack;
- Global sets (not used in MXF)

All MXF decoders must support local sets. Encoders should use the sets as required by the operational pattern. If there is no guidance in the operational pattern then the encoder should opt for a local set implementation using the local tags as defined in SMPTE 377M. Note that 2-byte lengths in local sets are always coded as big-endian (i.e., MSB first).

Every property in MXF has a full 16-byte Universal label so that the property may be interchanged with other systems as either a single KLV item or as a Universal set.

MXF-specified metadata is currently implemented using 2 byte tags and lengths. This restriction does not apply to private metadata schemes, although it is recommended because the primer pack mechanism for preventing numerical clashes of local tags, is only defined for two-byte tags.

5.7 Implementing text

Many of the text fields in MXF are encoded using UNICODE. The coding technique is UTF-16 with big-endian byte order to allow good international support. More information on UNICODE can be found in the reference in annex C.1. There are occasions when ISO-646 text is used. This is often to comply with some other standard such as the ISO-639 language descriptor codes.

Text is stored in a KLV or tag-length-value structure. Zero word termination of strings is optional. A string may be the same length as the “L” of the KLV or the “length” of the tag-length-value with no zero word at the end. Alternatively, a shorter string may be placed in the space allocated by the KLV or the tag-length-value structure by inserting a zero word after the last character of the string. MXF decoders must support both mechanisms.

5.8 Tracking changes with generation numbers

A generation number is a weak reference to the identification set that was created when the MXF file was saved or modified by an application. Each time the MXF File is modified, a new identification set is created. If a metadata set is changed, the generation ID property is updated so its value will be the same as the generation ID of the Identification Set that was created when the property was modified.

It is important to note that generation number properties are optional and that decoders should not rely on their existence; however, in certain applications, they can be very useful. If your application stores extended data that is dependent on data stored in AAF's built-in classes and properties, your application may need to check if another application has modified the data in the built-in classes and properties.

The generation property allows you to track whether another application has modified data in an MXF file that may invalidate data that your application has stored in extensions. The generation property is a weak reference to the identification object created when an MXF file is created or modified. If your application creates extended data that is dependent on data stored in MXF built-in classes or properties, you can use the generation property to check if another application has modified the MXF file since the time that your application set the extended data. To do this, your application stores the value of the generation UID of the identification object created when your application sets the value of the extended data.

6 Metadata classifications and placement

The main objective of the material exchange format is to exchange program material together with attached metadata information. This section provides a very brief overview of some of the underlying concepts of metadata as it is used within an MXF file. A fuller description of the use of MXF descriptive metadata can be found in the SMPTE EG 42.

In general terms, the use of metadata has many dimensions as follows:

1. It is in widespread use within different content-based industries, including broadcast, film, music and web authoring.
2. It is in widespread use in different content-based applications, including capture/creation, production, post-production and archive/libraries.
3. It can be divided into several different broad categories including business transactions, publication information, content identification and labelling, compositional information and formatting, etc.
4. It may have different states such as being static for a defined duration, being dynamic (with several kinds of dynamic including transitory, metronomic, incrementing and so on).
5. It may have different levels of stability with elements having durable values that remain stable or transient values that may frequently change.

Metadata can be divided into three broad categories:

1. Structural metadata: A set of information that defines the essence structure; i.e., how the essence was edited and what source components were included in what derivation chain.
2. Descriptive metadata: A set of information that describes, parameterizes or catalogues content, such as episode number, copyright holder and so on.
3. Dark metadata: Unknown to an application at the time of processing. This may be for many reasons including private metadata, unknown extensions to MXF and standardized metadata items that are not handled by the application.

MXF (and AAF) provide the ability to bind metadata, essence and data essence streams together via structural metadata. MXF also provides a descriptive metadata mechanism that allows independent DM schemes to be created as plugs into the overall MXF file format.

The placement of metadata in a file may be in one or more of several possible locations most suited to the application of the particular metadata item. Figure 10 indicates several broad locations where metadata may be stored.

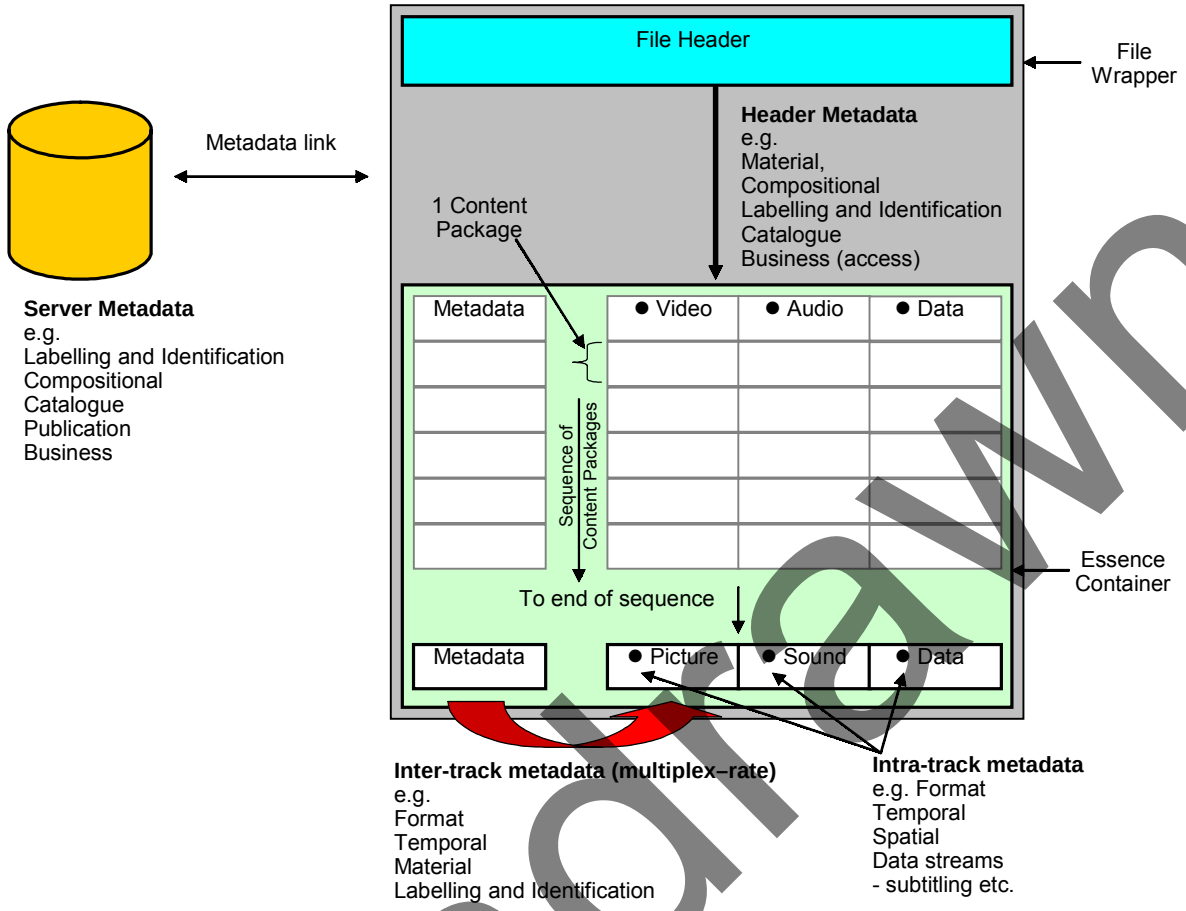


Figure 10 – Different locations for metadata storage

6.1 Embedded metadata location

Embedded metadata (intra-track in the figure above) is that which is tightly embedded in the essence stream such as is present in MPEG-2 Video ES and AES3 data. Metadata that is embedded is typically:

Format: for decoder operation;
Temporal: with particular reference to time-code;
Spatial: such as pan-scan vectors and aspect ratio;
Extra data: such as captioning, subtitles, etc.

6.2 Linked metadata location

Linked metadata (inter-track in the figure above) is that which is closely linked to the content, whether video, audio or data content, through a container on a picture-by-picture basis. Thus, this metadata is interleaved with the content and maintains a tight timing relationship with it. As an example, the system item of SDTI-CP provides this metadata location. Metadata that can be stored as linked to the frame is that relating to:

Format: often as a duplicate of the embedded metadata;
Temporal: mostly as temporally variable metadata extra to any embedded metadata;
Material: including the extended UMID; and
Label: simple labeling of the content.

6.3 Attached metadata location

Attached metadata (header metadata) is that which may appear in a file header such as is present in MXF. It can encompass a wide variety of metadata, in particular:

Content: providing metadata about the content in the file body;
Compositional: providing simple or complex editing information for the clip or program;
Label: providing a full set of content labeling and identification;
Catalogue: for location of events, markers and for archival metadata; and
Business: for access and security information.

6.4 Server metadata location

Server metadata can be used to replicate almost all of the metadata described so far. However, it is particularly useful for the following metadata sets:

Label: providing a full set of content labeling and identification metadata;
Compositional: providing simple or complex editing information and historical derivation metadata;
Catalogue: for use in off-line searches;
Publication: defining when and where content is to be delivered; and
Business: for audience information, program statistics, etc.

7 MXF in detail

7.1 General overview

SMPTE 377M defines a file format for the transfer of program material between equipment in the professional broadcast environment. Stream and file transfers are both used for the interchange of program material, with file transfers increasing in proportion to stream transfers. Neither will dominate; rather they will co-exist and the MXF file is designed to work within both transfer classes.

File transfer is different from stream transfer in several respects:

Files are often created directly from incoming streams and are often converted into streams for emission and distribution. The MXF standard specifies an MXF file format that is readily convertible to and from common streaming formats with low overhead and without loss of data.

In order to appreciate the differences between stream and file transfers, we can summarize the major characteristics of each as follows:

File transfers...

1. Can be made using removable file media;
2. Use a packet-based reliable network interconnect and are usually acknowledged;
3. Are usually transferred as a single unit (or as a known set of segments) with a predetermined start and end;
4. Are not normally synchronized to an external clock (during the transfer);
5. Are often point-to-point or point-to-multipoint with limited multipoint size;
6. File formats are often structured to allow access to essence data at random or widely distributed byte positions.

Stream transfers...

1. Use a data streaming interconnect and are usually unacknowledged;
2. Are open-ended, with no predetermined start or end;
3. Streams are normally synchronized to a clock or are asynchronous, with a specified minimum/maximum transfer rate;
4. Are often point-to-multipoint or broadcast;
5. Streaming formats are usually structured to allow access to essence data at sequential byte positions. Streaming decoders are always sequential.

Figure 11 illustrates the interoperation between streaming transfers based on stream interfaces such as SDTI and file transfers between disc servers and tape archives. One of the issues of the file transfer is that many servers support playout before file closure (i.e., read from a partially written file while it is still in the process of writing), so blurring the distinctions outlined above.

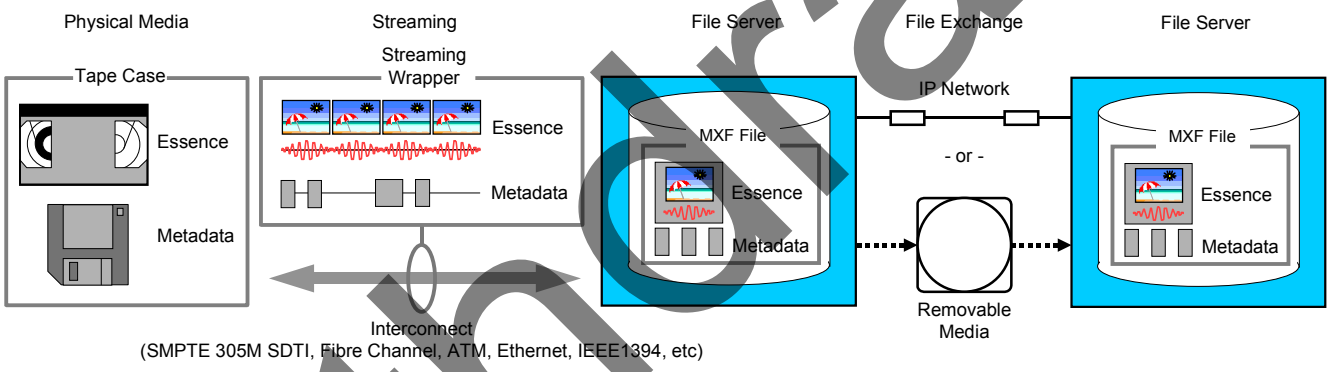


Figure 11 – MXF files and streaming formats

7.2 Content model

The content model used in SMPTE 377M is based on that defined by the EBU/SMPTE Task Force Report, which defines content as in figure 12.

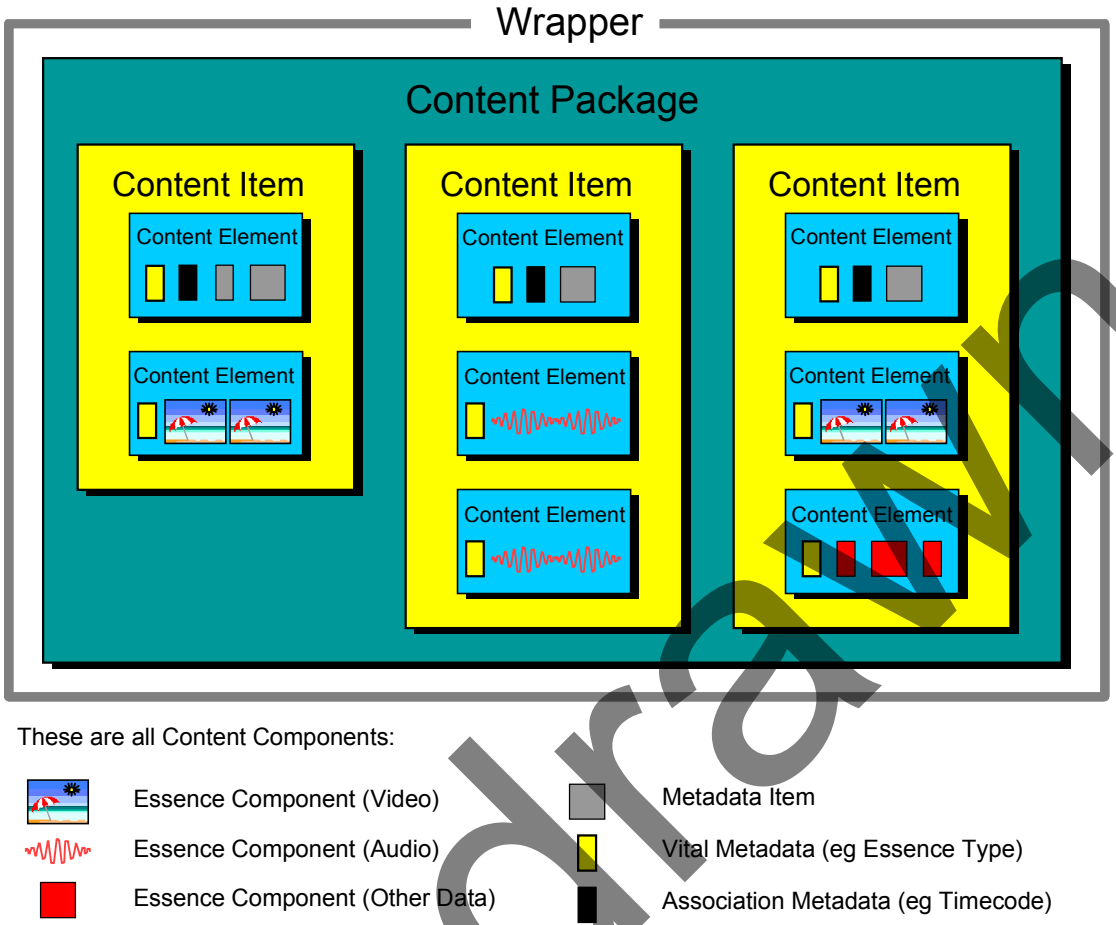


Figure 12 – Content package model

The content model also uses the terminology of SMPTE 336M (KLV coding) and SMPTE 298M (Universal labels), which define:

- Universal labels (ULs) used as keys;
- Key-length-value formatting of individual metadata and essence items;
- Coding of groups of data items into sets and packs.

The content model also uses the terminology of SMPTE 326M – SDTI-CP, which defines frame-interleaved content based on the following components:

System item: that includes system level descriptive metadata and content metadata;
Picture item: that includes one or more picture elements;
Sound item: that contains one or more audio elements;
Data item: that contains one or more data essence elements.
Compound item: that contains one or more intrinsically interleaved elements (such as an interleave of DV-DIF packets)
Link item: that links metadata in the system item to any one of the elements.

Each of these essence elements can be separately indexed in an index table and is also mapped to a track in the header metadata. The track is the metadata object that controls the way in which this essence element is used.

7.3 Operational patterns

Different applications produce and consume material of various degrees of complexity and structure, from a single clip to a multitude of clips and effects. Applications requiring only the simplest files should not be burdened with support of the most complex. To maximize interoperability MXF uses operational patterns to define constrained levels of file complexity.

During the development of MXF there were many different attempts at defining the functionality of an operational pattern. The goal was to create a number of axes that allowed software and hardware developers to create products with different levels of functionality (and hence cost). These different axes had to correspond to real world ways of working, and had to provide mechanisms for a file to be "flattened" from a complex operational pattern to a simple operational pattern in a way that made sense to someone working with the Multimedia content.

The description below is of the different axes followed by a non-exhaustive discussion of some applications.

7.3.1 Operational pattern "axes"

When trying to constrain the complexity of an MXF file, there are different axes or degrees of freedom that can be constrained independently. It is intended that the operational patterns be written and standardized as they are needed. Most operational patterns will be written as a constraint on the axes in this section. However, for certain specialized applications (such as allowing audio-only WAV files to be read by non-MXF devices) there may be specialized operational patterns that constrain the specification differently. Regardless of the operational pattern, any MXF decoder will be able to read the header and report the contents of the file and why it can or cannot process the file.

The operation pattern axes are arranged so that any operational pattern to the left, or above another operational pattern is a subset of its functionality. For example, operational pattern 3b is a superset of the functionality of OP1a, OP2a, OP1b, OP2b and OP3a, and includes not just the ability for each material package to access sequential top-level file packages, but also the ability to access a sequence of ganged top-level file packages.

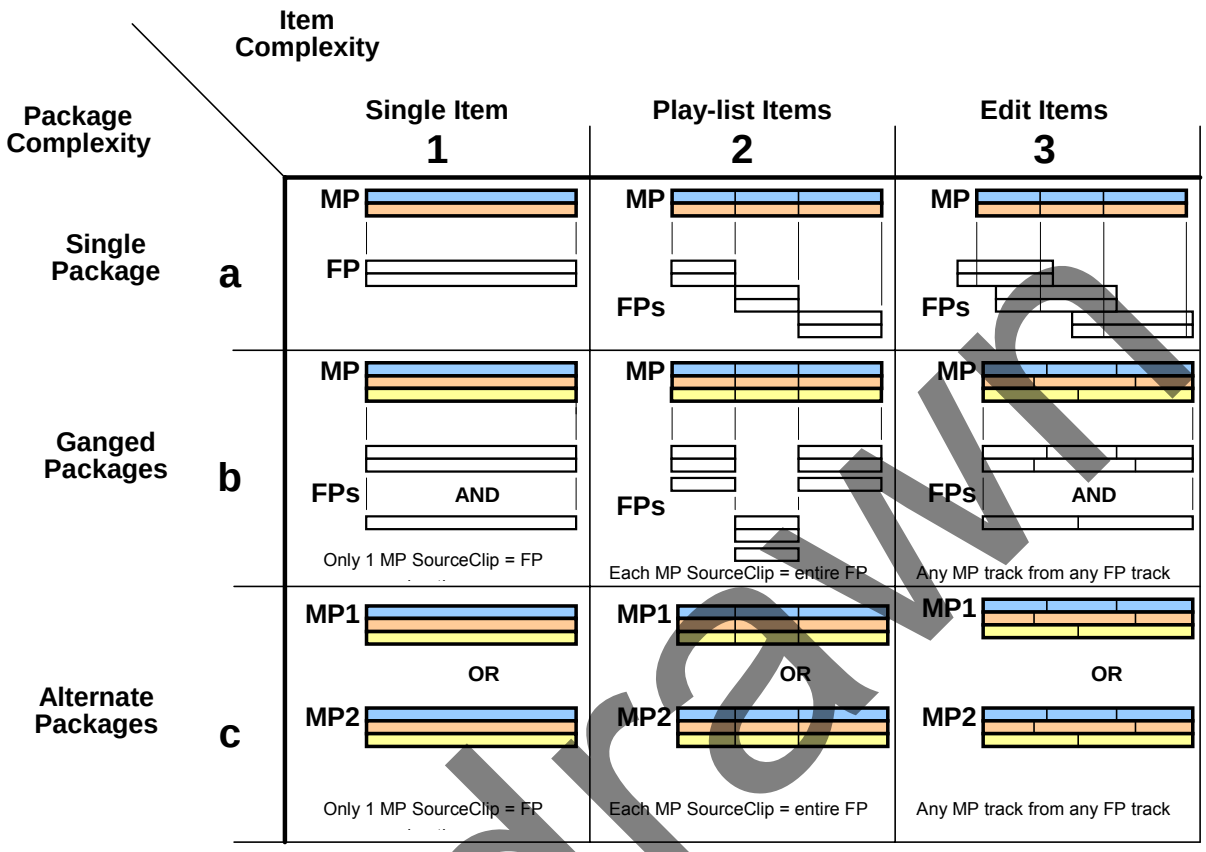


Figure 13 – Operational pattern axes

7.3.1.1 Item complexity

Here we constrain the temporal relationship between different top-level file packages within the MXF file. In principle, there are three levels of constraint:

1. **Single item:** The file contains top-level file packages that have the same duration as the output timeline (like a tape).

2. **Playlist items:** The file contains top-level file packages that are butted one against the other. All tracks are switched synchronously with optional audio fade out / fade in to prevent clicking. This can be likened to a playlist of tapes.

3. **Edit items:** The file contains several top-level file packages with one or more cut edits. Tracks may have independent editing to allow audio and video to be switched at different points in the timeline. This will often involve random access within the file and therefore MXF files in this column are unlikely to be streamable.

7.3.1.2 Package complexity

1. **Single package:** The file contains only one active essence container at any point on the output timeline.

2. **Ganged packages:** The file contains two or more essence containers that share a common synchronized timeline. The MXF structure is used to wrap several essence containers and multiplex them using the KLV and partitioning rules. This could be used to gang together an MPEG picture track in one package with an uncompressed sound track in another (possibly external) package.

3. **Alternate packages:** The file contains several versions of the "program". There are several material packages that might be used to control a browse track or different language versions of a program, or different edits of some finished material destined for different censorship zones. For example, an OP1c file may have two continuous timelines — one for the French soundtrack and another for the English soundtrack. Another example is an OP3c file, where not only is there a choice of English or French, but the cut lists for the output tracks are different. Since this OP is a superset of the ganged package complexity, it also has the capabilities of ganged packages as well as alternate packages.

7.3.2 Operational pattern qualifiers

In addition to the axes above, there are operational pattern qualifiers that modify the behaviors above.

7.3.2.1 Internal / external flag

This is a simple flag that modifies an operational pattern. It has two states to indicate that all the essence containers are internal to the file (internal) or that one or more of the essence containers are in an external file (eXternal). For example, an OP1bx file may have internal picture data, but external sound data. (🡻 8.2.7.1).

7.3.2.2 Stream / non-stream (wire / storage) flag

This is a simple flag that indicates either that the partitions in the file have been arranged so that it can be streamed on a wire (Wire file), or that some other non-streaming arrangement has been used (stored file). The streamed file representation implies that essence containers are multiplexed together and that within an essence container, any interleave that exists will allow decoding of the essence during streaming file transfer so that the pictures may be viewed and the sound heard during transfer with minimal latency. The size of buffers required to do this is an application issue and outside the scope of SMPTE 377M. Any file that does not have this property is just a file. (🡻 8.2.7.2).

7.3.2.3 Uni-track / multi-track flag

This is a simple flag that indicates that all the essence containers in an MXF file have only a single essence track. This flag is to aid workflows where all the different essence components of a production are required to be individual files. This flag helps MXF decoders know that the file meets this criterion. The flag is either uni-track or multi-track. (🡻 8.2.7.3).

7.3.3 Operational pattern applications

MXF applications should, where appropriate, be able to perform the following functions with respect to operational patterns:

Encoders and decoders should be able to report the most complex operational pattern they can handle.

A decoder should be able to indicate what level of operational pattern has been processed when its capabilities have been exceeded.

Encoders should always correctly signal the operation pattern of the files they create. This means that an MXF encoder capable of creating all possible operational patterns should not signal the files it creates with the highest operational pattern code. It should signal the operational pattern to which the file complies.

Listed below are several MXF applications and possible ways in which they may be implemented using SMPTE 377M. They are intended to give a guide on how MXF might be used. They are not normative definitions of the operational patterns concerned.

An application might give a file a name depending on its functionality, for example:

Test_OP1aiwm.mxf – mxf file with **i**nternal essence, **w**ire-file, **m**ultitrack, Operational Pattern **1a**;
Test_OP3cxm.mxf – mxf file with e**x**ternal essence, not streamable, **m**ultitrack, Operational Pattern **3c**.

7.3.3.1 Video tape replacement

A video tape is essentially a single container with a single item on it. Even though there may be more than one “scene” or “shot” or “clip” on the tape, no special processing is required to play the sequence. All the material is internal to the tape and it is stored in a way that can be streamed. This makes an operational pattern for video tape replacement one of the simplest operational patterns.

7.3.3.2 Archive

There are many different archive applications. Often, it is desirable to have metadata or a browse track “online” and the full-quality content in some deep store. This requires referencing of external essence as well as multiple representations of the same content. There may only be one single item in each of the representations (each having the same duration) and the content could be arranged for streaming or storage depending on the precise application.

7.3.3.3 D-Cinema

For distribution of D-cinema content, it may be desirable to have different representations of the same film distributed on common media. Alternatively, MXF may be used to represent each “reel”, which is then assembled via a composition list that itself may be an MXF File. Different representations may be as simple as different language tracks, or may be as complicated as different audio-video cuts to meet local or regional content restrictions. The operational pattern axes allow this split of functionality. In addition a D-cinema application will almost certainly require protection of the content. This can be achieved with a metadata plug-in to describe the encryption / protection scheme and an essence container type to contain the encrypted / protected essence(s). The other mechanisms within MXF remain unchanged.

7.3.3.4 Adding handles to material

Handles are extra bits of material before and after the desired content. There are several ways in which these could be implemented in MXF depending on the desired result.

The most common use of handles is to adjust edit points, and / or to provide context for production processes such as color correction. This use of handles implies that the content within the handle is not actually used in the material package, but exists within the top-level file package. The resulting file would be in the edit items column of the operational pattern axes matrix. The precise row or column of the operational pattern would depend on the construction of the essence within the file. For a mono-essence file, it would be constructed as an OP2a or OP3a file. Multi-track files would be either OP2b or OP3b depending on whether or not the cut points of the top-level file packages are synchronized on the timeline.

7.4 Relationship between MXF and essence containers

MXF files created in accordance with the MXF standard use essence containers to encapsulate one or more essence elements. These essence elements may be intrinsically interleaved (for example, a SMPTE 314M DV-based stream) or may consist of a single non-interleaved essence element.

In order to support stream capability, the essence elements are interleaved over a limited duration (typically 1 frame). Each essence element can be encapsulated using KLV coding over the interleave duration to allow an MXF decoder to access the essence on these KLV boundaries.
The MXF Format does not provide the individual essence container specifications, but defines the constraints that a compliant essence container specification must meet in order for it to be encapsulated in an MXF file body. Constraints on the essence container are given in the operational pattern document and the essence container document. They may be summarized as follows:

1. Must encapsulate each essence component with KLV coding using publicly registered keys;

2. Must provide for interleaving of the essence components over a limited duration (typically one frame), when inputs or outputs are use for streaming;

3. Must be standardized as an open specification, preferably through the due-process of SMPTE;

4. Must meet the SMPTE criteria for a standard (see the SMPTE Administrative Practices).

It is expected that compliant essence containers will become available for the systems below.

Note that none of the compression formats is a compulsory function.

7.4.1 MXF generic container

SMPTE 377M provides a generic container with intrinsic interleaving. This allows most existing formats to be mapped into the MXF Format with minimal invention of new techniques.

Wrapping all essence variants in a common essence container format is advantageous for system design and interoperability. The MXF document suite specifies mappings of a variety of essence formats into the MXF generic container as described below.

The MXF generic container may also use essence elements and metadata items defined in SMPTE 331M through application of the specifications in SMPTE 385M (mapping SDTI-CP essence and metadata into the MXF GC).

7.4.2 MPEG-2 long GOP and type D-10

MPEG compressed picture essence in streams may be interleaved in several different patterns as defined by the ISO 13818-1 systems layer, including elementary streams, program streams, and transport streams. SMPTE type D-10 MPEG elementary streams are defined by SMPTE 356M. An MXF essence container specification allowing wrapping of these essence types currently recommends frame by frame wrapping of elementary streams in the generic container as the preferred MXF encapsulation method.

7.4.3 DV compressed essence

MXF Files created in accordance with this specification are intended for use in systems employing the DV family of compression schemes defined by IEC61834-2, SMPTE 314M and SMPTE 370M.

7.4.4 Uncompressed pictures

MXF files may be used for the transfer of program material employing uncompressed video at all resolutions, including standard and high definitions. The MXF standards specify the use of the KLV data construct for encapsulating uncompressed video, and the use of a separate KLV packet to carry signal parameters for use by decoders and transcoders. Like all GC element mappings, this picture element may be used on its own, or may be used with appropriate sound or data essence elements.

7.4.5 Audio

An MXF mapping document for the encapsulation of AES3 audio and broadcast wave compatible audio in the generic container has been defined. This audio element may be used on its own, or may be used to add audio to another generic container element such as uncompressed pictures or MPEG long GOP pictures.

7.4.6 Other compression types

MXF Files may be used to encapsulate various other video essence compression systems, including M-JPEG, JPEG-2000, MPEG-4 simple, MPEG-4 studio profile, MPEG-4 part 10 video, and audio essence compression systems, including Dolby AC-3 and Dolby E.

7.4.7 Essence container and essence type identification

The types of essence permitted in each specific variant of MXF file are defined by individual essence container specifications and are identified in the file header by one or more unique essence container labels.

7.5 How MXF objects / sets relate to the essence container

SMPTE 377M is a physical representation of the underlying AAF class model and uses the same methods for data identification and data relationships. The method of relating the structural header metadata to the essence container is now described.

In each partition of an MXF file, there may be any or all of the following core components:

1. A partition pack that defines:
   – body SID for the container data stream in this partition;
   – an Index SID for the index table in this partition.

2. A primer pack.

3. Header metadata repetition that includes:
   – a content storage set at the top level;
   – one or more top-level file packages each associated with an essence container data set;
   – other metadata to describe the entire file (after all, it is a header metadata repetition).

4. An essence container (that occupies the whole file body or a part).

5. Unique IDs that link data sets together (16-byte Instance UIDs).

6. Unique material IDs (32-byte UMIDs) that identify the essence container.

These components are related as indicated in figure 14:

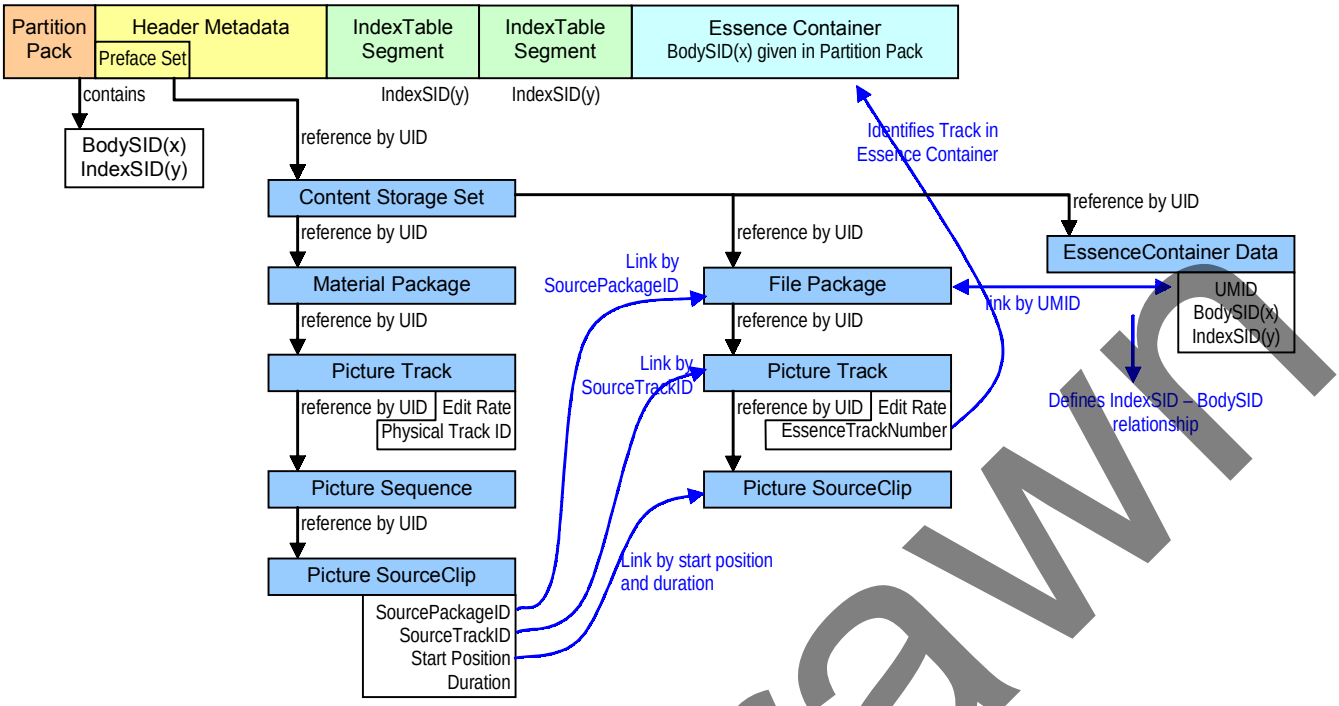


Figure 14 – MXF metadata and relationship to the essence container

The relationships are as follows:

The partition pack includes a BodySID and an IndexSID that identify the essence container segment and index table segments in the partition. These are linked to the BodySID and IndexSID in the relevant top-level file package via the corresponding EssenceContainerData set. They are also linked to the BodySID and IndexSID in the relevant index table. When the BodySID value in a partition is zero, it indicates that there is no essence container segment in this partition. Likewise, a zero IndexSID value indicates there are no index table segments in this partition.

The header metadata has a content storage set at the top level that contains a set of package UIDs and a set of EssenceContainerData UIDs. The content storage set strongly references every package, including each top-level file package as well as each material package. The content storage set will also reference lower-level source packages where these are present in the header metadata.

Within the header metadata, there is also an essence container data set for every top-level file package. This set provides the linking between BodySID, IndexSID and their related package UMID value. This mechanism relates the partitions and index tables within the file body to the top-level file packages in the header metadata.

NOTE – The package UIDs are basic UMIDs.

7.6 Discussion on endian-ism in MXF

The MXF format is intended to be platform neutral. This means it should not rely on resources available on any specific platform. There are, however, two distinct ways in which multi-byte numbers are stored in computer systems, big-endian and little-endian. Big-endian systems place higher value bytes in the lower value addresses, whereas little-endian systems do the reverse. This means that any data structure placed directly in a processor's memory by hardware can be read "in place" on one system, but must undergo a byte swap process in the other.

In addition, MXF is intended to have a common object model with AAF. AAF implements variable endian-ism based on a byte-order property within various classes.

Note that this feature applies only to the metadata elements in the file. The essence containers have fixed byte orders depending on the specification of the essence container.

There are several possible solutions in MXF, of which three are listed here:

1. **All header metadata items will be big-endian;**
2. **All header metadata items will be little-endian;**
3. **The MXF encoder will signal the endian-ness it used; i.e., source-endian.**

There were many design discussions during the development of MXF and the final conclusion was that MXF should be big-endian and should not indicate this in the file. The main reason behind this decision was to simplify the handling of dark metadata where the endian-ism cannot be known (because the metadata is dark).

7.7 MXF decoder design

MXF decoder design is, of course, an application-specific issue. This section is intended to advise implementers of issues that will improve interoperability with other systems. It is desirable that all MXF decoders should be able to **parse** (i.e., understand the syntactic structure) at least the following:

1. The KLV packet structure of all parts of the file (including the KLV packets of any kind of essence container);
2. The KLV structure of the header partition, any body partition and any the footer partition;
3. The KLV structure of any optional index tables;
4. The optional random index pack;
5. The basic header metadata structure in any partition;
6. Locate the SMPTE Universal labels in all the partition packs;
7. Skip over any run-in.

In addition, it is desirable that MXF decoders **decode** (i.e., interpret and act on the values within) at least:

The metadata sets and individual metadata items defined in the minimum implementation of the simplest operational pattern.

Decoding of other aspects such as the compressed bitstream or the specific essence container in the file body depends on the ability of the decoder to support those aspects. It is desirable that MXF decoders be able to locate and present the information that identifies the contents of the MXF file as follows:

1. The MXF file identification itself (that identifies that the file is MXF compliant) through the key value of the header partition pack;
2. The UL of the operational pattern (structural metadata) to which the file conforms;
3. An array of ULs that identify each essence container and its contents in the file body;
4. An array of ULs that identify each descriptive metadata collection within the file.

7.7.1 Minimum decoder concept

It may be useful to application specification writers to use the concept of a minimum decoder. This would have a defined functionality in addition to that listed above. Two examples are given below:

- The minimum decoder for a tape-based MXF player would include the ability to decode and unwrap the essence in a restricted number of compression types. It could include a “turbo” mode where aligning the data to a specified KAG value could guarantee faster-than-real-time behavior.

- The minimum decoder for a content-aware filing system would include the ability to determine which metadata sets were included in the file and to create menu items to allow the metadata schemes to be browsed. It may include thumbnail generation for a limited number of essence types. It might also enable database registration of the UMIDs and descriptive metadata with a media asset management system.

In general the minimum decoder will depend on the system in which the MXF file is being used.

7.8 External files — where is the essence

The MXF essence descriptor contains a list of properties called “locators”. MXF supports two different types of locator — network and text. The top-level file package that describes the essence (i.e., the one that is referenced by the material package) may have external essence, and the decoder must scan the locators *in the order they are given* to find the essence. A typical example of this might be the creation of a CD-ROM where the network locators are given as a file reference relative the location of the MXF file, followed by other locations in which the file might be found; e.g.:

Network locator: “src/clip1.dv” — a relative file reference to clip1.dv in folder **src**;
Network locator: file://usr/~jon/clip1.dv — an absolute file reference to clip1.dv in **jon**’s home folder;
Text locator: “clip1 DV tape is on shelf 42” — a text locator intended for a human to interpret.

Even though the actual essence data is external to the file, there may be metadata describing the essence within the file. In the extreme case, all the essence could be external to the file leaving a small MXF stub that fully describes the external essence. MXF Files with Internal essence may also have locators. When all the essence can be found internally, the locators should be treated as being for information purposes. In higher operational patterns, it is possible that some of the essence will be internal and some of it will be external. In this case, internal essence, where present, should take precedence over external references. Where there is no internal essence available from a material package SourceClip reference, the locators should be searched in their listed order to find the content (see also 8.2.7.1). External content can be verified by checking the BodySID value in the essence container set for the appropriate UMID. A zero value indicates external essence.

8 MXF worked examples

8.1 Identifying the contents of an MXF file

This section is written in a decoder-centric fashion to illustrate why certain parameters are stored the way they are. An encoder should create a file so that the maximum number of decoders is likely to be able to read / decode it. What does this mean? In practice, it means that the MXF encoder’s designers may discover that there are choices to be made when creating MXF files. It may be the case that “elegant little tricks” with the MXF syntax are found that may make life easier for the encoder designer. If the use of such tricks reduces the chance of interoperability with simple decoders, these tricks should be avoided. MXF is an interchange file format and the goal of all MXF devices should be to maximize the probability of Interoperability.

The order in which an MXF device or application searches for parameters within the file depends very much on what the device or application is trying to do with the file. For example:

- An MXF file explorer GUI probably wants ownership information from the identification set;
- An MXF asset manager needs to know UMIDs of the current and previous versions as well as whether the content is in the file or externally referenced;
- An MXF tape device probably wants the size of the header metadata and the essence container type;
- A computer based MXF playback application probably wants to know the operational pattern and what essence container type(s) are in the file;
- An MXF edit conformer needs to know the essence container types and whether or not all the essence is Internal to the file.

Notice from the list above that there are valid and important MXF applications that do not need to know the exact essence type and are never likely to decode the content. To be able to read the file, the MXF decoder is likely to go through a number of steps in both the physical and logical structures of the file.

8.1.1 Is it an MXF file?

All MXF files start with an optional run-in followed by the header partition pack key. The run-in is less than 64k bytes and the condition for finding the start of the file is to identify the first 11 bytes of the partition pack key. The simplest way to do this is to scan the initial 64k bytes of a file for these 11 bytes. When they are found, the MXF specific decoding can begin

8.1.2 Is this an MXF File that my application can process?

MXF has been designed to allow the generation of "early failure" messages. This means that MXF Decoder designers should attempt to determine as early as possible whether or not they can wholly or partially process a given MXF File. Where possible, feedback should be given to the user if the application is not able to process some or all of the file. Typical reasons might be:

- "No codec available for essence container type <name>", where <name> is the human- readable (in the local language) name of the essence container as determined by a dictionary;
- "Unknown essence container type <number> - not found in database", where <number> is the UL of the essence type that cannot be handled because it was not found in the local dictionary;
- "Operational pattern complexity exceeded. This file is OPxx, this device can play files of complexity OPyy".

It is crucial that MXF encoders create files with accurate header information. An MXF encoder may be asked to create files that are simpler than the highest operational pattern it was designed to create. It is a normative provision of the specification that the MXF encoder correctly set its header information. For example, if an MXF encoder can create files of OP1b complexity, but is asked to create a file with a single mono-essence top-level file package, then the MXF encoder must signal "OP1a" complexity in the header.

Most of the "fail fast" information required by a decoder can be found in the partition pack. Typical processing by the decoder may be:

- *Is this an MXF version I understand?* The MXF decoder checks the MajorVersion and MinorVersion properties of the partition pack and checks them against the decoder's reference value. Note that in future versions of SMPTE 377M the partition pack key may have differences in bytes 14, 15 and 16 compared to previous versions of the specification.
- *Is this an operational pattern I can handle?* The MXF decoder checks the operational pattern UL against the list of ULs it knows how to handle.
- *Is the data in this partition stable?* The MXF decoder checks byte 15 of the partition pack key to determine if this partition is of type "closed" or "closed and complete". If the partition is of type "open", then the MXF application should find another partition pack because the information in this one may have been created on the fly and may be inaccurate.
- *Can I decode or process the Essence?* The MXF decoder processes the EssenceContainers batch in the partition pack to compare each label against a list of labels it knows how to process. It is possible that the essence will be stored in several essence containers of the same type (e.g., 3 DV clips) — in this case, there will be only one instance of the EssenceContainer label. It is also possible that there will be a single EssenceContainer in the file and that this will contain several different interleaved essence types — for example, there may be uncompressed images in a generic container interleaved with several tracks of AES audio. In this case, there would be two essence container labels — one for the uncompressed pictures and the other for the interleaved audio.

- *What is the duration of the file?* The MXF decoder searches for the primary package UID in the preface set and discovers the duration by inspecting the duration property of the sequences of the tracks in that package.
- *What device made it?* This information is stored in the identification set which can be found using the most recent generation UUID.
- *Is it HDTV or SDTV?* This can be determined by inspecting the essence descriptor for the picture track. The picture track in the top-level file package(s) has a property called TrackID. This will match one of the linked TrackID values in one of the EssenceDescriptors within the file. This EssenceDescriptor contains many properties that fully describe the source picture essence. These include horizontal and vertical sizes as well as the frame rate and nominal aspect ratio of the content.
- *Where is the external essence?* Each essence descriptor has a Locators property, which is an ordered list of places where the essence might be. This list should be searched in order to find the essence. A locator may be a URL or it may be text intended for a human operator (e.g., "all known URLs have been searched (<list of URLs inserted by application>) and the essence was not found — it came from the green cassette on the shelf behind the water cooler"). Mechanisms for finding external essence are outside the scope of this document, but media asset management systems that use UMIDs for identification are becoming more common at the time of writing of this document.

8.2 Partitioning a file

8.2.1 Partitioning for streaming — the streamable file

When streaming an MXF file, it is desirable to reduce the size of the buffers needed in the receiver, which in turn reduces the overall latency of the system. To be streamable, a file will usually contain an interleave of picture and sound elements. In many systems that use compressed sound material, it is likely that the smallest unit of sound does not have the same duration as the field or frame duration of the pictures. The guidelines below are intended to improve the chances of interchange when streaming and refer to the placement of elements in the content package of the MXF generic container. The term "access unit" is borrowed from MPEG to indicate the smallest unit of content that can be allocated a time value. Figure 15 shows the basic structure of a content package — different essence items that each contain different essence elements. The Items can appear in any order, but all elements of the same type must be contiguous.

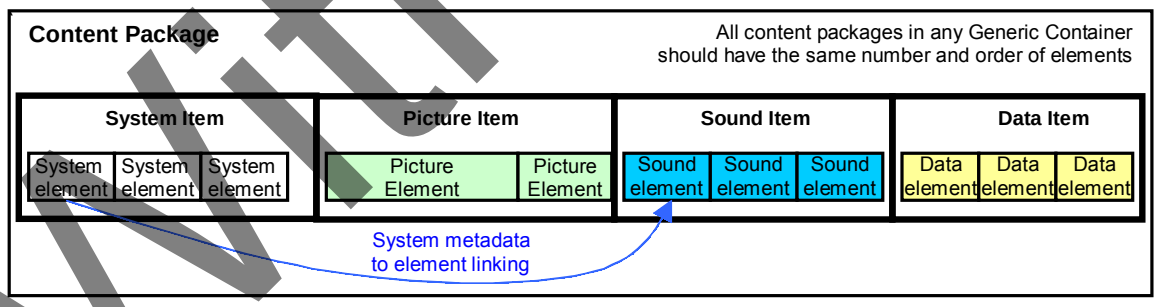


Figure 15 – Logical structure of items and elements in a content package

In each content package:

1. There is one picture access unit.
2. The synchronized sound sample should be in the first sound element in the same content package. This implies that the start position of the picture access unit should be equal to the start position of the sound element or fall within the duration of the first sound element.

3. Sound elements should be placed in the content package until a sound element is found that may start a later content package. (Note that when the sound element duration is greater than the picture access unit, this results in content packages with no or zero length sound elements).
4. Any data element should start with the first indivisible unit of data where the start position of the video access unit is equal to the start position of the data element or falls within the duration of the first data element.
5. Any data element should end with the unit of data whose position on the timeline is not later that the position of the next video access unit.

These guidelines create files that are streamable, but may require large receiver buffers to synchronize the picture, sound and data. Many compression specifications provide a lot of information on buffering and streaming, and creating a system with similar buffer characteristics is the goal here. For example, the MPEG-2 specification ISO/IEC 13818-1 gives rules and guidelines for multiplexing the audio and video streams into either a program stream or a transport stream.

When streaming a file, the decoder is intended to display the pictures and recreate the sound while the file is being sent. The delay through the video and audio decoders is often not the same; therefore buffering is required in the decoder to bring the sound and pictures into synchronization. This buffering is often in addition to any buffering required for compression decoding and basic demultiplexing of the streams.

The guidance given here is that an MXF encoder should create a stream as though it were creating the content for streaming using the underlying compression standard; the GC content package guidelines above should then be applied. This should result in a good compromise between low latency and KLV decodability.

8.2.1.1 OP1a file requirements

This simple operational pattern is the one that is most likely to be used for streaming. This operational pattern normatively requires that "*… the essence container shall provide for the continuous decoding of contiguous essence elements with no processing. The essence container or essence element specifications may add extra restrictions to this condition*".

This constraint is to ensure the continuous decodability of the essence. It does not constrain changes in aspect ratio, active format descriptor, colorimetry or any other parameter that can vary without resetting or crashing an essence decoder. Changes of picture size, frame rate, essence coding mode, discontinuities in timing parameters and errored data are all examples of essence decodability conditions that would break the OP1a requirement. It is important to note that even if the OP1a essence decodability conditions are met, the file must still be wrapped and delivered in an appropriate fashion to be a streaming file.

8.2.2 How do I know what sort of track or package I've got?

Each package in an MXF file has an array of strong references to tracks. Following these references will give the track sets that describe the content for this package. A material package can be identified by its key value and will have no essence descriptors. Top-level file packages and lower-level source packages will have essence descriptors. For mono-essence content, the descriptor will either be a type of file descriptor or a type of physical descriptor or, for other essence, a multiple descriptor. The file or physical source package type can be determined as follows:

1. If there is one descriptor and it is a file descriptor, then the package is a file package.
2. If there is one descriptor and it is a physical descriptor, then the package is a physical package.
3. If there is a multiple descriptor and **any** of the descriptors referenced by the multiple descriptor's array are physical descriptors, then the package is a physical package.
4. If there is a multiple descriptor and **all** of the descriptors referenced by the multiple descriptor's array are file descriptors, then the package is a file package.

The primary package property of the preface set indicates which package is to be considered the primary package. For an MXF player application, this is the package that should be played out by default. For an MXF

Ingest application, the primary package is the one that most accurately describes the Ingested material. By default, this will be the material package of the file.

Now that the underlying package type is known, the relationships between the packages can be determined. The material package, or material packages, have tracks that have sequences that have SourceClips that refer to top-level file package tracks. Only these top-level file packages are allowed to describe actual essence. The top-level file packages have tracks that have sequences that have SourceClips that may reference lower-level source packages. These lower-level source packages contain historical derivation information. Lower-level source packages, whether file packages or physical packages, will always describe essence that is external to the MXF file.

Now that all the packages are known, the track types need to be identified. In MXF, all tracks look the same and it is not until the sequence referenced by the track is inspected that the track type is known. Similarly, all sequence sets look the same and it is not until the data definition property value is resolved that the track type can finally be worked out. The values of the ULs corresponding to the different track types are given in the SMPTE labels registry. There are different UL values for picture, sound and data tracks; this data definition value should be consistent between the sequences and SourceClips along a track as well as those up and down the source reference chain.

8.2.3 How multi-top-level file package files are arranged

As mentioned already in this guideline, the logical and physical representations of a file are essentially orthogonal. Any generalized operational pattern MXF file that is not OP1a will have multiple top-level file packages. The physical arrangement of the essence described by these top-level file packages will depend on the qualifier bits as well as the operational pattern.

The most obvious physical constraint is to make a file that is streamable (🡻8.2.1). When there are multiple top-level file packages in the file, managing streaming buffers becomes slightly more complicated because of the requirement that the essence for each top-level file package must be in a partition with a unique BodySID value. The management of the data in the partition packs and any index table segments must be done in such a way that the receiver essence buffers are still kept in a condition that prevents overflow and underflow.

8.2.3.1 Which top-level file package goes with which material package track?

Each material package SourceClip has two properties that identify the appropriate top-level file package:

SourcePackageID — a 32-byte basic UMID
SourceTrackID — a 4-byte Uint32 track Identifier

These identify respectively the top-level source package and the track within it. The referenced top-level file package set will have a PackageUID property that is the same as the SourcePackageID property of the material package SourceClip. This top-level file package will have an InstanceUID that is in the batch of strong references to packages in the ContentStorage set (when the top-level file package is stored within the file).

8.2.3.2 Which partition of essence goes with which top-level file package?

The important parameter here is the BodySID value, which is found in one of the essence container data sets. Having identified the top-level file package UMID, which was the same as the SourcePackageID in the material package SourceClip, each of the essence container data sets is searched until the package UID is found in the linked package UID property. This set will contain a BodySID value and an IndexSID value that are used to identify the partitions that contain the essence data and index table data for this top-level file package. This BodySID value will be found in the BodySID property of the partition packs where essence data can be found.

8.2.3.3 Which index table goes with which essence container?

The IndexSID value found in the matched essence container data set will be found in the IndexSID property of the partition packs where index table segments can be found. According to the partitioning rules and the index table rules, there is a unique index table for each of the top-level file packages. This unique index table will contain segments that will only be found in partitions where the IndexSID has the correct value.

8.2.3.4 Which KLV wrapped essence goes with which track?

This section is only relevant if the essence container has interleaved picture, sound, data or systems elements. Each of the interleaved elements within the identified partition must be associated with a track in order for MXF to describe them. The track number property of the track set is used to identify the essence within the essence container.

For essence containers that use the MXF generic container, the track number property will match bytes 13-16 of the key of wrapped essence data. Specific details of these 4 bytes can be found in the MXF generic container specification as well as the individual generic container mapping documents.

8.2.3.5 Which part of the top-level file package do I use?

The initial answer to this question seems easy — it is the part referenced by the material package SourceClip. Here are the steps taken to resolve the reference including some finer points of the specification that are sometimes overlooked:

1. The material package SourceClip has a SourcePackageID (UMID) property that identifies the top-level file package.
2. The material package SourceClip has a SourceTrackID that identifies the TrackID of the track within the top-level file package that is to be used.
3. The material package SourceClip has a StartPosition property that determines the start point along the track in the top-level file package.
4. The material package SourceClip has a duration property that determines how long the clip lasts.

Assuming the edit rate of the material package track is the same as that of the top-level file package. Assuming also, that both tracks have origin values of 0, it is straightforward to determine which portion of the essence to use.

If these assumptions are not valid, some math is required to determine the correct start point. In SMPTE 377M, synchronization is discussed in section 8.4. The equation for synchronization is copied below:

Essence on tracks *n* and *m* are synchronized when: $\frac{Position_n}{EditRate_n} = \frac{Position_m}{EditRate_m}$

In addition, a SourceClips StartPosition is measured in edit units of the track containing the SourceClip, **not** of the referenced track. This means that when material is redigitized or relinked, you don't have to go and renormalize all the tracks that reference that material.

Now it should be clear that the desired position along the referenced track (in edit units of the referenced track) is given by the equation below:

Position along File Package Track is $Position_{fp} = EditRate_{fp} \times \left(\frac{Position_{mp}}{EditRate_{mp}} \right)$

However, this is not the end. The origin parameter for the file package indicates how much stored essence exists before the position=0 point on the track. The final equation giving the start point along the stored essence measured in file package edit units is therefore given by the equation below:

$$Offset\_From\_Stored\_Essence\_Start_{fp} = (Position_{fp} + Origin_{fp}) = EditRate_{fp} \times \left(\frac{Position_{mp}}{EditRate_{mp}} \right) + Origin_{fp}$$

8.2.4 Creating a file with multiple top-level file packages

When a file with multiple top-level file packages is not being streamed, there may be no constraints governing the construction of the file. Under these circumstances, this guideline recommends that each of the different essence containers within the file is kept contiguous within the file — even when each essence container is segmented into multiple partitions.

The next question to be answered is “In which order should the essence containers appear in the file?”

If it is known that some of the essence containers are more likely to be changed than the others (for example audio tracks that might be edited), then those essence containers should occur last in the file. The essence container that is least likely to be changed should be placed first in the file.

If no knowledge of the likelihood of change is available to the MXF encoder then the essence containers should be ordered so that the largest essence container appears first in the file. There are always going to be circumstances when this rule is not optimal (e.g., when preview pictures are in the file), so implementers are advised to think carefully about application requirements before committing to firm multiplexing rules.

8.2.5 Creating a file with multiple material packages

In many ways, a file with multiple material packages is simpler than one with multiple top-level file packages. There is no extra essence to be added, only extra metadata to give a choice of different timelines using the content within the file. A few simple examples of this may be:

- OP1c – single picture track with a choice of different language sound tracks
- OP1c – single picture track with a choice of stereo / multi-channel sound tracks
- OP1c – choice of lo-res preview pictures with mono sound or hi-res pictures with multi-channel sound.
- OP2c – feature material with a choice of languages on the sound tracks and selectable language specific Picture clips at the start of the feature material
- OP3c – feature material that has selectable clips (or reels, or whatever terminology is used) within the feature for localization of the feature.

In general, the arrangement of the essence within the file should follow the same rules as a file in rows a or b of the operational pattern axes matrix. If a file is marked as streamable, then this means that each and every material package is streamable. If a file is marked as having internal essence, this means that all the essence for all the file packages is internal. The essence described by the top-level file packages must follow the guidelines section 8.2.4 and any interleaving guidelines (e.g., streaming guidelines in section 8.2.1) that exist for the essence type being used.

The question of “which material package do I use” is an application-specific question, but in general the package whose Instance UID value appears in the preface pack’s primary package property should by the one chosen if no additional information is available.

8.2.6 Achieving robustness for file recovery and partial restore

One of the design requirements of MXF was to accommodate partial restore and provide file transfer robustness. The design feature to implement both of these applications is the use of partitions. It has already been noted in this document that a partition pack may be inserted at the beginning, end or anywhere in the middle of the file. It is these body partitions in the middle of the file and the use of the random index pack that allow file recovery and partial restore.

8.2.6.1 File recovery

This application can be split roughly into two different scenarios:

1. **A push-mode file transfer was interrupted or joined after the start;**
2. **A stored file needs checking for consistency.**

In both of these cases, partition packs need to be inserted regularly and frequently enough for the physical parameters to allow recovery without the loss of too much data. How much is too much? Well, that is a highly application-specific question and may be as small as a frame, or as big as the entire file. For this reason, an MXF encoder targeted at this sort of application must be designed with an awareness of the data loss that could arise from the partition spacing that is used.

The partition pack has two properties that should be consistent throughout the file:

ThisPartition: The offset to the start of this partition in the sequence of partitions (as a **byte count** relative to the start of the header partition);

PreviousPartition: The offset to the start of the previous partition in the sequence of partitions (as a **byte count** byte relative to the start of the header partition).

In addition, the start of an MXF file is identified by the first 11 bytes of the key of the partition pack.

It should now be possible to see that a push-mode transfer may be joined halfway through the stream by detecting the first 11 bytes of a partition pack. If this is a valid partition pack then the remaining byte of the key will match a known partition pack, and the values within the pack will contain valid values. The very first partition of the file is always the header partition and will have a “ThisPartition” value of 0. If a push-mode transfer is joined and “ThisPartition” is non-zero, then the number of missed bytes can be determined.

The PreviousPartition value can be used as a rough measure of the rate of insertion of partitions (assuming that there is some consistency to the partitioning strategy used by the MXF encoder). It should also be noted that although the first 11 bytes of the partition pack key is quite a long byte sequence, it is not necessarily sufficiently unique to never occur in the essence of a file. For this reason, a more robust decoder may wait until the second partition pack header is received and check that:

$$ThisPartition_n - PreviousPartition_n = ThisPartition_{n-1}$$

Checking a stored file for consistency now involves counting bytes within a file and verifying that all the ThisPartition and PreviousPartition properties are correct.

8.2.6.2 Partial restore

This application is subtly different from the one above. The application needs to extract a recoverable portion of the (possibly damaged) original file and present it as a new MXF file.

Files that act as the master for this operation should be constructed with regular body partitions, a random index pack (RIP) and index tables. Ideally a complete index table for each and every essence container will exist both in the header and in the footer of the file.

The portion of the file to be extracted will most often be expressed in terms of time along the file. This example will only consider the case of an operational pattern 1a file. In the higher operational patterns, extra work must be undertaken to ensure that the correct portions of each and every referenced top-level file package are extracted. The complexity of the index table handling will also increase because there is one index table per essence container that may be segmented. Each essence container must be handled separately with the RIP being used to identify the start of each partition.

In any MXF file, a RIP can be detected by accessing the last 32 bytes of the MXF File and using this as a Uint32 backwards offset from the end of the file to the start of the RIP (precise details are in SMPTE 377M). If the RIP is present then the offset will point to the first byte of the KLV key of the RIP. The RIP can now be read and the start point of each of the partitions in the file can be determined. In an OP1a file, this data is less critical than in a higher operational pattern file where the partitions will also be used to separate the different essence containers. In OP1a files, there is only one essence container and therefore only one index table. An index table segment can now be located by finding a partition with the correct IndexSID value in the partition pack.

Now that the index table has been found, the byte offsets within the essence stream can be found by an index table look-up. If the partial file extraction is to be done with a minimum of processing, then all the partitions from the one containing the first byte up to and including the last partition containing the last byte can be extracted.

It is strongly recommended that after this extraction process has been done, the partition header data be processed to correct the MXF file:

- The “ThisPartition” and “PreviousPartition” values in each partition header should be corrected;
- Index tables should be created that are consistent with the new partial file;
- The UMIDs should be updated to show that this is not the same as the original material. (A combination of SMPTE RP 205 and operational practice will determine the exact UMID modification required).

8.2.7 Setting the operational pattern qualifier bits

The MXF operational pattern has three qualifier bits that provide global information about the internal arrangement of the data within the file. This section is intended to clarify how these bits should be set and to explain some of the pathological cases that may not otherwise be clear.

8.2.7.1 Bit 1: Internal / external essence

At first glance, this seems obvious — either the content is internal or it isn’t. MXF allows referencing of external essence containers via locators in the top-level file package. However, locators are allowed to be present even when there are essence containers internal to the file. This implies that bit 1 should be zero **only** if all top-level file packages in the file have matching essence container data sets and essence containers in this file.

Are locators the only way of finding external metadata? No. If a material package references a file package that is simply not present in the file, then this is a valid external reference. In this case, bit 1 would have to be set. Finding the essence is rather more difficult — an external media asset management system needs to be used in order to resolve the UMID and find the content.

The next three figures attempt to show three different conditions that could result in external essence. Figure 16 shows linkage using only UMID as the linking mechanism. The material package contains a SourceClip with a SourcePackageID (UMID) that is not in the file. This can be determined by inspection of all the top-level file packages and optionally by the presence of an essence container data set with a BodySID of 0. Some

external mechanism (such as an asset management system) is required to resolve this UMID to a filename that can be inspected for a UMID match as shown in the lower part of the diagram.

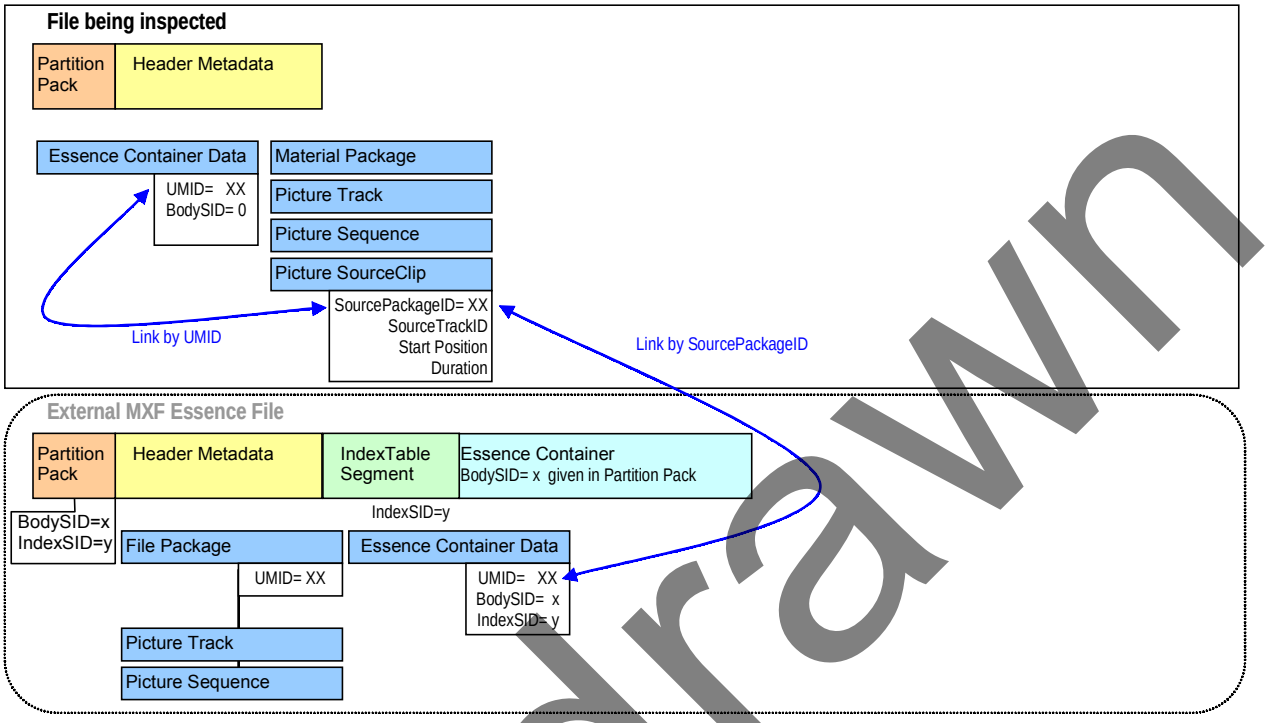


Figure 16 – External essence example using only UMID for linking

Locators provide a mechanism for discovering the location of external essence using only information within the file. The advantage is that no external mechanism is required; the disadvantage is that when the external file is moved, the locators should be updated. Figure 17 shows a similar example to the one above, although this time the material package contains a SourceClip with a SourcePackageID (UMID) that appears to be in the file. Why “appears to be”? Because a file package exists in the file with the correct UMID, but the essence container data set indicates the BodySID value is 0. There are, however, two network locators and a text locator. The first of these text locators is resolved to the file in the lower half of the figure. The locator identifies non-MXF essence and because of this, it may be difficult for an application to check the UMID for correctness.

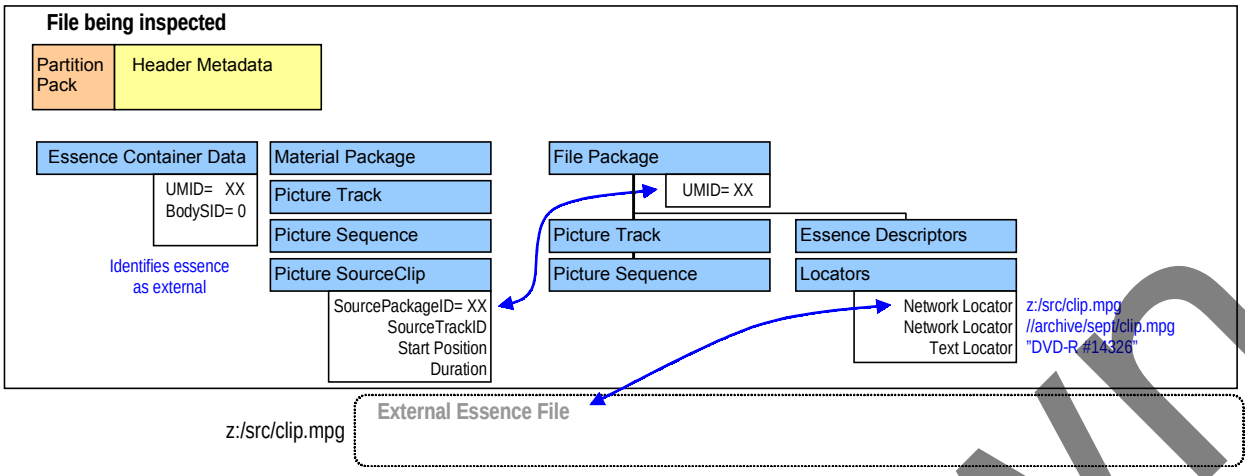


Figure 17 – External essence example using locators for linking

Figure 18 shows an example where the external essence is an MXF file. As in the examples above, a material package references a file package that appears to be in the file. The essence container data set indicates that the essence is external because the BodySID is 0, equally obviously because there is no essence in the file! The locator resolves to an MXF file, and this time checks can be made to determine that the target of the reference is correct. The top-level file package of the target file will have the same values as the top-level file package in the first file. If the UMIDs match, the target file has been found. If not, the rest of the locators should be inspected as above.

The top-level file package in the external file should be identical to that in the original file. If there are any discrepancies, then the metadata values in the external file should take precedence. The top-level file package in the original file should be regarded as a copy.

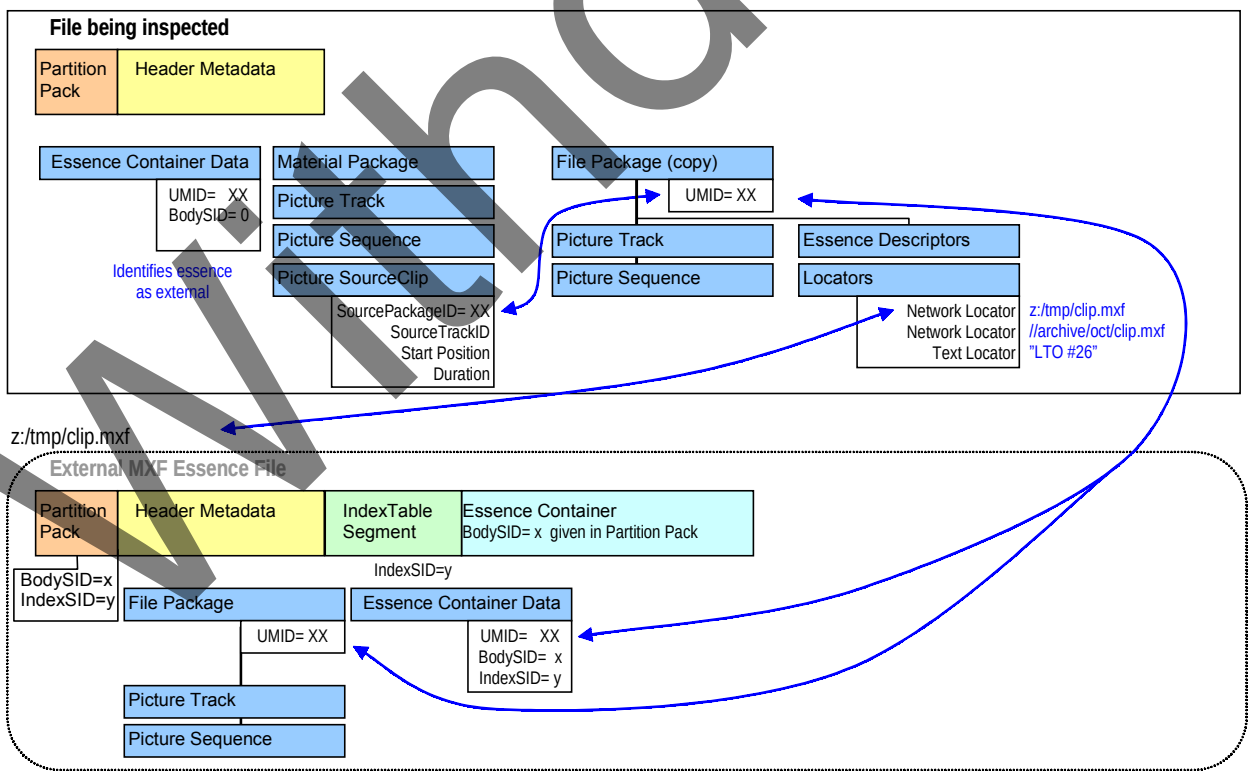


Figure 18 – External essence example using locators and UMIDs for linking

8.2.7.2 Bit 2: Stream file / non-stream file

The best description of this is found in MXF format specification in section 9.2: "The Essence Containers used in streaming operational patterns must be capable of interleave over a defined interleaving period or must be capable of being multiplexed in an MXF file using the partition mechanism. The interleave / multiplex duration is dependent upon the application, but should be the period of the minimum duration of usable picture essence, typically a picture frame period."

Reading the paragraph above two or three times, it seems clear. One possibly ambiguous case is where a file contains only a single essence container that is intrinsically streamable but is clip-wrapped, either in a generic container or in its own native container. In this case, bit 2 should be set to "**W**ire File" because the resulting file is still streamable according to the definition above. The interleave duration is set by the intrinsic streamability of the underlying essence and there is no partitioning (i.e., multiplexing). The application has determined, therefore, that the multiplex duration is equal to the length of the file.

Some cases of streamable status are clear and unambiguous. However, other cases can be subjective. The following illustrate some possible cases of streamable files (all assuming that the essence is, itself, streamable):

1. A single, frame-wrapped, EC with a single essence element (e.g., OP1a with B-Wav essence).
2. A single, frame-wrapped, EC with multiple interleaved essence elements (e.g., OP1a with type D-10 mapping).
3. Multiple, frame-wrapped, ECs where the ECs are in presentation sequence and in contiguous partitions (e.g., OP2a with type D-11 mapping).
4. A single, clip-wrapped, EC with a single essence element (e.g., OP1a with MPEG-2 long-GOP video ES).
5. A single, clip-wrapped, EC with an inherently interleaved essence stream (e.g., OP1a with a DV DIF stream).
6. Multiple, clip-wrapped, essence elements, each in separate ECs, which are multiplexed over clips of short duration (say <1sec) (e.g., OP2b with MPEG-2 I-frame video ES multiplexed with B-Wav audio).

8.2.7.3 Bit 3: Uni-track / multi-track

At first glance, this too seems straightforward — there is either one track or more than one track. The definition of this bit refers to the essence container, not to the file package. This bit is intended to give the intention of the essence container, for example a stereo AES file should be signaled as a single track because the left and right channels are treated together. A DV essence container in which only the video is used should be flagged as uni-track because that is the intention of the essence container.

Note that the goal of this flag is to describe a uni-track file; i.e., an OP2a file could be uni-track because it could be constructed to have only one active track in the output timeline. OP1b could not be uni-track because there will always be two or may synchronized tracks active on the output timeline.

8.2.7.4 Higher operational patterns

In operational patterns higher than OP1a, care needs to be taken when setting these qualifier bits. An example is an OP1b that is split into two parts (MP = material package, FP = top-level file package):

Part1 = MP + FP pictures (internal) + FP sound (external);
Part2 = MP + FP pictures (external) + FP sound (internal).

The following conditions are true and would result in setting of qualifier bits:

- Each file is uni-track so bit 3 is set to uni-track;
- Both were constructed to be streamable (as a result of the way they were created) so bit 2 set to "wire file";
- Each file has external essence so bit 1 should be set to "external".

If the primary package property is set be the internal top-level file package of the file, then it is possible that the file may be played, even though it is flagged as having external essence.

8.3 Index tables

MXF index tables are intended to be versatile, compression agnostic, streamable and applicable to any and all of the MXF operational patterns defined in SMPTE 377M. The purpose of an Index Table is to convert from time offsets to byte offsets within a file. The MXF Index Table specification may, at first seem rather complex, but its resulting versatility gives huge functionality to random access systems:

- Cameras and streaming devices can create segmented index tables on the fly;
- Storage devices may have index tables at the start, end or both;
- Index tables are created for each essence container. Multiplexing essence containers or changing the partitioning of a file does not change the index table.

The index table structure for an essence container is defined by the “delta entries”. There is one delta entry for each of the interleaved elements of the essence container. These delta entries allow an essence element to be categorized as either CBE (constant bytes per element) or VBE (variable bytes per element). MXF encoders should always “play it safe” if there is any uncertainty that an element is CBE. Each and every element in the entire file should have the CBE byte count — if this is not true then each index table must use the slice mechanism to indicate a VBE stream.

8.3.1 Using index table delta entries

Figure 19 shows the physical and logical representations of a content package for use in this index table example. The physical content package shows an edit unit *n* with five elements and some fill at the end of the content package. The logical arrangement of these elements is shown on the right hand side of the picture.

In this example, there are three separate essence tracks that need to be indexed. The data and sound elements are all CBE, but the interleaving rules used for this essence container lead to a variable number of sound elements per content package. In the content package shown in figure 19, there are two sound elements. This results in a VBE sound stream for the purposes of Indexing because the number of bytes of sound data for a given edit unit is not constant.

The MXF index table delta entries are intended to allow identification of each of the indexed elements, and to indicate whether they are CBE or VBE elements. We will partially fill in the DeltaEntry array here with the CBE values we know. The VBE values will be filled in once slices have been introduced.

In table 4, the expression BC_{System} indicates the byte count for the system element.

Table 4 – An MXF index table delta entry array without slices

	Item Name	Meaning	Value	Why …
	NDE	Number of delta entries	5	There are 5 delta entries as shown below
	Length	Length of each delta entry	6	Each one is 6 bytes long
Delta Entry 0 System	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry	0	It's the start of the Index Table – slice 0
	Element Delta	Delta from start of slice to this Element	0	It's the first entry – offset 0bytes from the start of the start of this Indexed Edit Unit
Delta Entry 1 Data	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry	0	The previous element was CBE, so this is still slice 0
	Element Delta	Delta from start of slice to this Element	BC_{System}	This element starts at the end of the System Element, so this value is the byte count of the System Element
Delta Entry 2 Picture	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry		
	Element Delta	Delta from start of slice to this Element		
Delta Entry 3 Sound	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry		
	Element Delta	Delta from start of slice to this Element		
Delta Entry 4 Fill	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry		
	Element Delta	Delta from start of slice to this Element		

8.3.2 Using index table slices

If every entry of an index table were CBE, the slice mechanism would not be needed. The size of each element would be known and the DeltaEntry array would be filled in as above. In our example, we have three variable length elements — the picture, sound and fill. The index table is "sliced" so that a VBE element is the last element in a slice. In our example, slice 0 is terminated by the picture element, slice 1 is terminated by the sound Element, and the final slice is terminated by the fill element. We can now fill in the slice information in table 5:

Table 5 – An MXF index table delta array with slices

	Item Name	Meaning	Value	Why …
	NDE	Number of delta entries	5	There are 5 delta entries as shown below
	Length	Length of each delta entry	6	Each one is 6 bytes long
Delta Entry 0 System	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry	0	It's the start of the Index Table – slice 0
	Element Delta	Delta from start of slice to this Element	0	It's the first entry – offset 0bytes from the start of the start of this Indexed Edit Unit
Delta Entry 1 Data	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry	0	The previous element was CBE, so this is still slice 0
	Element Delta	Delta from start of slice to this Element	Sizeof(System)	This element starts at the end of the System Element, so this value is the byte count of the System Element
Delta Entry 2 Picture	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry	0	This is the Element that terminates slice 0
	Element Delta	Delta from start of slice to this Element	BC_{System} + BC_{Data}	The offset to the start of the Picture item is the byte count of the system Element + the byte count of the Data element
Delta Entry 3 Sound	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry	1	This is the Element that terminates slice 1
	Element Delta	Delta from start of slice to this Element	0	It is also the first element in slice 1
Delta Entry 4 Fill	PosTableIndex	Temporal Reordering / Index into PosTable		
	Slice	Slice number in IndexEntry	2	This is the Element that terminates slice 2
	Element Delta	Delta from start of slice to this Element	0	It is also the first element in slice 2

The use of the PostableIndex field is given in section 8.3.5.

8.3.3 Indexing frame-wrapped and clip-wrapped essence containers

In frame wrapping mode the tables index the first byte of the key that wraps the indexed frame. This will most likely be the first byte of the picture element key or sound element key in the appropriate GC mapping specification. This means that each element delta values include the lengths of the "KL" for each element.

If the overall length of all the elements in each frame is constant, then a delta entry array and an "edit unit byte count" item are sufficient to define the index table segment.

In clip wrapping mode, the tables index the first byte of the data for each indexed frame. For example, in the MPEG long GOP case, this will be the first byte of the start_code for the appropriate access unit. This means that each element delta values give precisely the length of the data for each frame.

If the overall length of all the data for each and every indexed stream is constant for all frames, then a delta entry array and an "edit unit byte count" item are sufficient to define the index table segment.

8.3.4 Fixed sized essence

This simple example comes from the generic container DV mapping document.

This is a simple case where the index table points to the first byte of the DV-DIF compound element generic container key. There are no other generic container items and any sound or data information is embedded within the DV-DIF container. Therefore, the only pieces of information in the index table segment are the start position, duration and (fixed) size of each DV-DIF compound item KLV triplet.

Table 6 – Frame wrapped index table segment set

Item Name	Req ?	Meaning	Use
Index Table Segment	Req	An Index Table Segment set	See MXF Format Specification
Length	Req	Set Length	See MXF Format Specification
Instance ID	Req	Unique ID of this instance	See MXF Format Specification
Index Edit Rate	Req	Edit Rate copied from the tracks of the Essence Container	See MXF Format Specification
Index Start Position	Req	The first editable unit indexed by this Index Table segment measured in File Package Edit Units	Set to the position value of the first edit unit indexed by this Index Table segment.
Index Duration	Req	Time duration of this table segment measured in Edit Units of the referenced Package	May be set to zero to indicate that this Index Table Segment applies to all Edit Units in this Essence Container
Edit Unit Byte Count	D/Req	Defines the byte count of each and every Edit Unit. A value of 0 defines the byte count of Edit Units is only given in the Index Entry Array	Set to the number of bytes in every KLV, including the length of the Key and Length. The Index Table can be used to find the first byte of the KLV of every DV-DIF frame
IndexSID	D/Req	Stream Identifier (SID) of Index Table	See MXF Format Specification
BodySID	Req	Stream Identifier (SID) of the indexed Essence Container	See MXF Format Specification

8.3.5 Variable sized essence

This worked example comes from the MPEG mapping document. It is intended to show the construction of an index table for a frame wrapped interleave of sound, data and picture elements where it is important to preserve the synchronization of the picture, sound and data to an accuracy of better than a frame.

This is a case where the index table points to the first byte of the MPEG picture element generic container key. The other generic container elements should be indexed by correct use of the delta entries and index entries. This example assumes that the sound elements that are Indexed require the use of the fractional position mechanism defined in SMPTE 377M. The following figure represents a typical content package being indexed. This figure is based on a figure in SMPTE 377M.

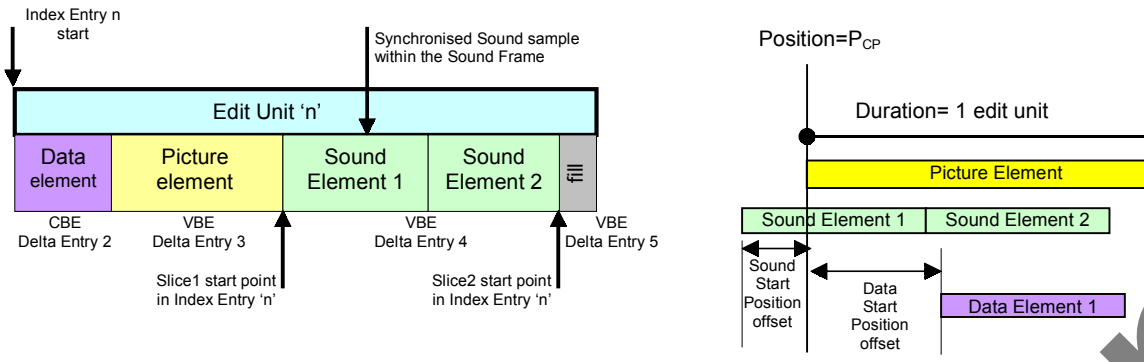


Figure 19 – Content package forindex table example

In this example, the picture, sound and fill are all VBE. The fill is indexed so that it can be eliminated from any essence byte counting based solely on calculations in the index table.

Table 7 – Frame wrapped index table segment set example

Item Name	Req ?	Meaning	Use
Index Table Segment	Req	An Index Table Segment set	See MXF Format Specification
Length	Req	Set Length	See MXF Format Specification
Instance ID	Req	Unique ID of this instance	See MXF Format Specification
Edit Rate	Req	Edit Rate copied from the tracks of the Essence Container	See MXF Format Specification
Start Position	Req	The first editable unit indexed by this Index Table segment measured in File Package Edit Units	Calculate
Duration	Req	Time duration of this table segment measured in Edit Units of the referenced Package	Calculate
Edit Unit Byte Count	D/Req	Defines the byte count of each and every Edit Unit. A value of 0 defines the byte count of Edit Units is only given in the Index Entry Array	0 unless the total length of all the GC Elements are of constant size. In this example for we assume that only the Data Elements are VBR so the value is 0.
IndexSID	D/Req	Stream Identifier (SID) of Index Table	See MXF Format Specification
BodySID	Req	Stream Identifier (SID) of the indexed Essence Container	See MXF Format Specification
Slice Count	D/Req	Number of slices minus 1 (NSL)	2
PosTableCount	Opt	Number of PosTable Entries minus 1 NPE	1
Delta Entry Array	Opt	Map Elements onto Slices	Table 8
Index Entry Array	D/Req	Index from Edit Unit number to stream offset	Table 9

The delta entry array contains an entry for every indexed element in the generic container. The order of the elements in the delta entry array matches the order of the elements in the generic container. The example below is a delta entry array designed to match the example in figure 19. Implementations should construct a delta entry array according to the properties of the actual essence in the file.

In this example, we have several variable length elements so that an index entry array is required.

Note also that the delta entry array does not distinguish which element is which in the index table. To know which element is indexed, the following rules apply when an MPEG long GOP stream is indexed:

- Each content package starts with the same number and order of elements and the previous content package.
- If new elements are introduced for whatever reason, they are appended to the end of the existing content package elements.
- If elements in the content package have no data, then an IndexEntry for a zero length VBE element is created.
- Index tables have the same number of delta entries as the maximum number of elements in any content package.
- The essence type of an index entry can be determined by inspecting the key that wraps the indexed essence.

Table 8 – Frame wrapped delta entry array example for figure 19

	Field Name	Type	Meaning	Use
	NDE	UInt32	Number of delta entries	4
	Length	UInt32	Length of each delta entry	6
Data Delta Entry	PosTableIndex	Int8	0= No reordering +ve = PosTable Index	1 (1st PosTable Entry)
	Slice	UInt8	Slice number in IndexEntry	0
	Element Delta	UInt32	Delta from start of slice to this Element	0
Picture Delta Entry	PosTableIndex	Int8	-ve - reordered	-1 (reordered Long GOP content)
	Slice	UInt8	Slice number in IndexEntry	0
	Element Delta	UInt32	Delta from start of slice to this Element	sizeof(KL) + sizeof(Data)
Sound Delta Entry	PosTableIndex	Int8	0= No reordering +ve = PosTable Index	2 (2nd PosTable Entry)
	Slice	UInt8	Slice number in IndexEntry	1
	Element Delta	UInt32	Delta from start of slice to this Element	0
Fill Delta Entry	PosTableIndex	Int8	0= No reordering +ve = PosTable Index	0
	Slice	UInt8	Slice number in IndexEntry	2
	Element Delta	UInt32	Delta from start of slice to this Element	0

Table 9 – Frame wrapped index entry array description for figure 19

	N	Field Name	Type	Meaning	Use
	1	NIE	UInt32	Number of index entries	=number of frames
	1	Length	UInt32	Length of each index array entry	calculate
One Index Entry for every frame	N I E	Temporal Offset	Int8	Offset in edit units from Display Order to Coded Order	0
		Key-Frame Offset	Int8	Offset in edit units to previous Key Frame. The value is zero if this is a Key-Frame.	0
		Flags	EditUnitFlag	Flags for this Edit Unit Bit 7: Random Access Bit 6: Sequence Header Bit 5: forward prediction Bit 4: backward prediction 00= I , 10= P, 01 or 11= B Bits 0-3: reserved	calculate
		Stream Offset	UInt64	Offset in bytes from the first KLV element in this Edit Unit within the Essence Container Stream	Offset from the first byte of the key of the KLV for the first frame to the first byte of the Key of the KLV for the Data Element in this frame as shown in Figure 19
		SliceOffset	NSL x UInt32	The offset in bytes from the Stream Offset to the start of this slice.	Optional depending on the complexity of the VBR items. In this case there are 3 slices and NSL is set to 2
		PosTable	NPE *Rational	The fractional position offset from the start of the Content Package to the synchronized sample in the Content Package	This should be calculated for each wrapped element to ensure precise synchronization is maintained. There are 2 elements requiring offsets in this example so NPE=2

The table above shows the descriptions of the various elements required in the index entry. Below, the table shows entries for the first six frames of a long GOP sequence. The following values have been used in creating the table:

- The GOP display sequence for frames 0-5 is $B_0I_1B_2P_3B_4P_5$. This is the indexed order of the frames.
- The GOP transmission order for frames 0-5 is $I_1B_0P_3B_2P_5B_4$. This is the stored order of the frames.
- The GOP is closed (i.e., the first B frame contains predictions only from the I frame).
- The data element length is fixed at 700 bytes and temporally offset by –0.25 edit units
- The I frames are 48000 bytes, P frames are 9000 bytes and B frames 1000 bytes.
- In the six content packages, there are eight sound elements.
- The number of sound elements are multiplexed in the content packages as follows: (1)(1)(2)(1)(1)(2).
- Each sound element is 1000 bytes.
- Each fill element is 300 bytes.

Table 10 – Frame wrapped index entry array example

	N	Field Name	Type	Value	Note
		NIE	UInt32	6	6 Entries in this Index Table Segment
		Length	UInt32	35	Sizeof(IndexEntry) including the SliceOffsets & PosTable
0	Index Entry[0] – "B"	Temporal Offset	Int8	1	B_0 is stored in Content Package[1]
		Key-Frame Offset	Int8	1	The key frame is IndexEntry[1] and 1-0=1
		Flags	EditUnitFlag	D0h	Closed GOP B frame – backward reference, sequence_header & random access
		Stream Offset	UInt64	0	Offset of the 1st Stored CP in the Stream – I_1
		SliceOffset[0]	UInt32	48700	sizeof(data) + sizeof(I_1 frame)
		SliceOffset[1]	UInt32	49700	sizeof(data) + sizeof(I_1 frame) + sizeof(sound)
		PosTable[0]	Rational	0	Temporal offset from start of data to start of video
		PosTable[1]	Rational	0	Temporal offset from start of sound to start of video
1	Index Entry[1] – "I"	Temporal Offset	Int8	-1	I_1 is stored in Content Package[0]
		Key-Frame Offset	Int8	0	The key frame is IndexEntry[1] and 1-1=0
		Flags	EditUnitFlag	00h	I frame – no reference
		Stream Offset	UInt64	50 000	Offset of the 2nd Stored CP in the Stream – B_0
		SliceOffset[0]	UInt32	1700	sizeof(data) + sizeof(B_0 frame)
		SliceOffset[1]	UInt32	2700	sizeof(data) + sizeof(B_0 frame) + sizeof(sound)
		PosTable[0]	Rational	-1/4	Temporal offset from start of data to start of video
		PosTable[1]	Rational	-1/3	Temporal offset from start of sound to start of video
2	Index Entry[2] – "B"	Temporal Offset	Int8	1	B_2 is stored in Content Package[3]
		Key-Frame Offset	Int8	-1	The key frame is IndexEntry[1] and 1-2= -1
		Flags	EditUnitFlag	30h	B frame – bidirectional reference
		Stream Offset	UInt64	53 000	Offset of the 3rd Stored CP in the Stream – P_3
		SliceOffset[0]	UInt32	9700	sizeof(data) + sizeof(P_3 frame)
		SliceOffset[1]	UInt32	11700	sizeof(data) + sizeof(P_3 frame) + 2*sizeof(sound)
		PosTable[0]	Rational	-1/4	Temporal offset from start of data to start of video
		PosTable[1]	Rational	-2/3	Temporal offset from start of sound to start of video
3	Index Entry[3] – "P"	Temporal Offset	Int8	-1	P_3 is stored in Content Package[2]
		Key-Frame Offset	Int8	-2	The key frame is IndexEntry[1] and 1-3= -2
		Flags	EditUnitFlag	40h	B frame – forward reference
		Stream Offset	UInt64	65 000	Offset of the 4th Stored CP in the Stream – B_2
		SliceOffset[0]	UInt32	1700	sizeof(data) + sizeof(B_2 frame)
		SliceOffset[1]	UInt32	2700	sizeof(data) + sizeof(B_2 frame) + sizeof(sound)
		PosTable[0]	Rational	-1/4	Temporal offset from start of data to start of video
		PosTable[1]	Rational	0	Temporal offset from start of sound to start of video

	N	Field Name	Type	Value	Note
4	Index Entry[4] – "B"	Temporal Offset	Int8	1	B_4 is stored in Content Package[5]
		Key-Frame Offset	Int8	-3	The key frame is IndexEntry[1] and 1-4= -3
		Flags	EditUnitFlag	30h	B frame – bidirectional reference
		Stream Offset	UInt64	68 000	Offset of the 5th Stored CP in the Stream – P_5
		SliceOffset[0]	UInt32	9700	sizeof(data) + sizeof(P_5 frame)
		SliceOffset[1]	UInt32	10700	sizeof(data) + sizeof(P_5 frame) + sizeof(sound)
		PosTable[0]	Rational	-1/4	Temporal offset from start of data to start of video
		PosTable[1]	Rational	-1/3	Temporal offset from start of sound to start of video
5	Index Entry[5] – "P"	Temporal Offset	Int8	-1	P_5 is stored in Content Package[4]
		Key-Frame Offset	Int8	-4	The key frame is IndexEntry[1] and 1-5= -4
		Flags	EditUnitFlag	40h	B frame – forward reference
		Stream Offset	UInt64	79 000	Offset of the 6th Stored CP in the Stream – B_4
		SliceOffset[0]	UInt32	1700	sizeof(data) + sizeof(B_4 frame)
		SliceOffset[1]	UInt32	3700	sizeof(data) + sizeof(B_4 frame) + 2*sizeof(sound)
		PosTable[0]	Rational	-1/4	Temporal offset from start of data to start of video
		PosTable[1]	Rational	-2/3	Temporal offset from start of sound to start of video

8.3.6 Audio only index tables

In an audio only index table the choice of Indexing frequency is at the discretion of the application designer. Many audio formats have a fixed number of bits per second, and it may be desirable to use the constant bytes per element mechanism as outline in section 8.3.3.

Where there are a variable number of bytes per element, the IndexEntry mechanism needs to be used as shown in the examples above. An appropriate indexing rate is often to provide one Index entry per second.

8.3.7 External essence index table

When an MXF index table is used to index external content, the problem of reliability of the data comes into play. When the index table was created, the data was accurate. At some later date, when the information is used, an MXF application must make some basic checks to be sure that the index table applies to the file being processed (e.g., verify file length vs. max (index table byte offsets), and perhaps check that a few random entries point to frame start points.)

When indexing an external essence container, it is recommended that index tables be constructed in the same way they would be constructed if the essence container were internal. When indexing external data that is not KLV wrapped, the index table should be created where the byte offsets refer to the first byte of each edit unit of the essence — as in clip wrapping. Typically this will be the first byte of each frame.

8.4 Wrapping essence in the generic container

8.4.1 Long GOP MPEG with uncompressed audio and other data

Long GOP MPEG is unlike many other essence types in that video frames are reordered when stored. This leads to complications in the creation of index tables and the synchronization of associated audio and data

elements. The MXF MPEG mapping document goes into detail of how the different elements should be arranged to achieve synchronization and to improve interoperability.

This MXF engineering guideline recommends that frame wrapping of long GOP MPEG should be used wherever possible. It also recommends that the interleaving guidelines should be followed so that the relationship between the essence elements in each content package is consistent.

The interleaving rules are designed so that when a group of content packages are extracted, the likelihood of extracting the synchronized picture, sound and data elements is maximized. The figure below shows the physical arrangement of the KLV triplets in a file. It can be seen that the different channels of sound and data are KLV wrapped and kept contiguous with the picture KLV.

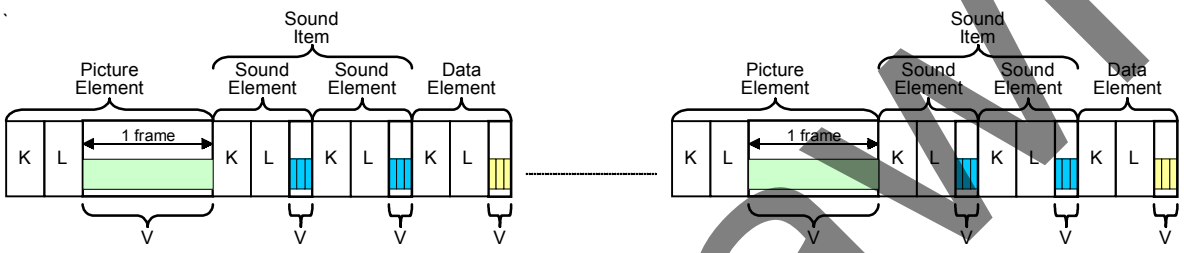


Figure 20 – Frame wrapping with other GC elements

8.4.2 Uncompressed video and audio

Although uncompressed video tends to generate the largest file sizes, the physical structure of the files is simpler than long GOP MPEG. There is no temporal reordering of the frames in an uncompressed video file. Figure 20 is also applicable to the frame wrapping of uncompressed video with associated audio. To know the exact format of the uncompressed video, the essence descriptor must be inspected. Specifically, the PixelLayout property defines the storage format of each of the pixels that collectively comprise the stored Image. The figure in annex E of the format specification defines many of the Image parameters, but the PixelLayout parameter deserves further mention here:

The intention of PixelLayout is to provide an algorithmic way of expressing the stored pixels of bit packing schemes which are likely to be used. The PixelLayout property is a zero terminated pairing of character codes and Uint8 bit depth values. These are all defined in SMPTE 377M, but a brief example illustrates the principle.

To describe 8-bit component 4:2:2 pixels packed into a 32-bit word, the 601 sequence would be:

Cb, Y, Cr, Y, Cb, Y, Cr, Cb, Y, Cr …

If these bytes were stored contiguously in an MXF file, the PixelLayout property to describe this arrangement would be:

PixelLayout= { ‘U’, 8, ‘Y’, 8, ‘V’, 8, ‘Y’, 8, 0, 0 }

This decodes as: 8 bit U (Cb) followed by 8 bits Y, followed by 8 bit V (Cr) followed by 8 bits Y. The final 2 zero values terminate the property.

8.4.3 VBI data

SMPTE 331M specifies the formats of VBI lines and ancillary data to be used in SDTI-CP systems. SMPTE 385M defines the mapping of such CP essence elements to the MXF generic container including the definitions of the KLV construct. It is recommended that these definitions be used for the carriage of VBI line

data and ancillary data (both H-ANC and V-ANC). Note that Anc packets have a data type identification that identifies the payload in the Anc packet. VBI data is, by its nature, untyped and can carry any kind of payload without any local identification.

8.4.4 Wrapping private essence

Wrapping private essence types involves the creation of unique values within the MXF metadata that ensure that the essence is consistently described during an MXF file interchange. Ideally the data values that are used should be SMPTE Universal labels and the essence wrapping would eventually be standardized. This will not be the case for all essence types however. This section of the guideline will assume that the package, track and essence identification sections above are well understood.

It is recommended that all data be wrapped using the MXF generic container specification — even private data. This allows the maximum reuse of existing tools, tests, code and knowledge in the MXF interchange environment. This example will assume a mapping of private essence to the generic container.

To identify the private essence, there are certain unique identifiers that need to be generated:

1. Keys to wrap the private essence;
2. An essence container UL to describe the essence containment used;
3. An essence descriptor with appropriate ULs to describe the actual essence;
4. A data definition for use in the sequences and SourceClips.

When the generic container is used, the first of 13 bytes of the key are already defined.

An essence container UL should ideally be a registered SMPTE UL. This could probably be an organizationally registered UL if the essence type is regularly used by an organization. Mechanisms for registering within SMPTE are being created as this document is being written.

8.5 Adding metadata

8.5.1 Adding private metadata

Private metadata additions to MXF can fall into two broad categories:

1. **Extensions to existing MXF sets;**
2. **New KLV coded sets and groups.**

The first of these uses a mechanism called the primer pack to prevent number clashes in the 2-byte tag values within the set. Essentially the 2-byte tag value is shorthand for the full 16-byte Universal label that identifies the set. The primer pack is essentially a list of all the 2-byte tags used in the file and their 16-byte equivalents.

A private metadata item should have a unique 16-byte identifier. It is recommended that any metadata Item that is likely to be used often is registered with SMPTE and a 16-byte UL is allocated. This is not always possible, so a 16-byte UUID may be generated instead. It is important that this UUID is understood both by the encoder and decoder of this private data, otherwise the data cannot be interchanged.

To extend an existing MXF set, the MXF encoder places the 16-byte identifier for the data in the primer pack and generates a 2-byte local tag from the “dynamic” range of numbers given in the format specification. It is important to check that this allocated number is not already used within the primer pack of the file.

Once this procedure is complete, the private metadata value can be added to the appropriate local sets in the MXF specification. The primer pack mechanism ensures that all decoders that don’t recognize the 16-byte identifier will ignore it. In order to respect the AAF data model, all private metadata additions to the MXF specification must follow the requirements of the single-inheritance hierarchy rules of the AAF class hierarchy.

Failure to follow this rule may lead to decoder errors. The best way to add private metadata such that it is compatible with the AAF data model is to study the model, which is available from the AAF Association (see annex C.1).

A new KLV coded set or group is more straightforward to add. A new 16-byte UL must be registered with SMPTE to wrap the set. The set should use the primer pack mechanism outlined already if 2-byte tags are being used in the set, otherwise the new set should follow SMPTE 336M. This set may be specified so that it does not need to follow the single inheritance hierarchy rules and may therefore be “dark” to AAF decoders.

At the time of writing this document, a private metadata carrier set was being designed. If the design is successful, this will provide a standardized way of adding sets of private metadata items to the MXF specification.

8.5.1.1 Intimate metadata (e.g., aspect ratio and AFD information)

Intimate metadata carries properties that are intimately associated with the essence. The metadata is usually dynamic in nature and may have as many bytes as the essence itself. When the intimate metadata is very large; e.g., 3-D depth map information, the metadata should be treated as essence with the guidelines for private essence being followed.

If the intimate metadata is quite compact, it may be appropriate to represent it as private metadata in the header. An example could be the representation of camera movements as private events on a descriptive metadata track.

It is likely that well-known intimate metadata properties such as aspect ratio and AFD information will have intimate metadata mechanisms defined for tracking them in MXF.

It is also likely that certain kinds of metadata may be carried within the essence container itself, such as in some elements of a system item of the generic container.

8.6 Working with time code

In MXF, time code is metadata annotation. The concept of time in MXF corresponds to a number of edit units along a particular track. To determine the time code at a given position on a track, the value of the time code segment must be calculated or read for that time code track. It is highly recommended that all MXF files are created with time code tracks in the material and top-level file packages, although this is not a normative requirement. MXF decoders should still operate correctly if the time code track is missing.

8.7 Playing a file backwards

Parsing an MXF file in the forward direction is a relatively simple task thanks to KLV coding. Parsing the file in the backwards direction is much more difficult without help. At the time of writing this guideline, a proposal exists for a simple generic container system element that does nothing more than provide a backwards pointer to the previous KLV wrapped content package. This allows very simple devices to provide forwards and backwards play.

8.8 Creating new descriptive metadata (DM) plug-ins

This subject is covered in great depth in SMPTE EG 42 and will only be lightly covered here. The plug-in mechanism is very simple and has the features described in the next sections.

8.8.1 Temporal properties of the metadata

MXF provides three sorts of track. These define whether the content on the track is static —(static track (DM)), may have overlapping or discontinuous content — (event track (DM)) or must have continuous content

with no overlaps — (timeline track (DM)). These track types may have metadata content placed on them according to the properties of the metadata.

8.8.2 Inserting metadata values or referencing metadata values?

MXF provides a DM segment for relating metadata values to one of the tracks. The DM segment contains the MXF properties that allow the descriptive metadata to fit into the MXF model along with a StrongRef to a DM framework, which is the “socket” into which a new descriptive metadata plug-in plugs.

MXF also provides a DM SourceClip for referencing descriptive metadata. This is useful in the case where an application wants to say, “The descriptive metadata for the top-level file package is the same as the lower-level source package”. Rather than duplicating the metadata values, a reference can be created between the two packages.

8.8.3 Linking properties of the metadata to tracks

Not all descriptive metadata applies to all the tracks. For example in an arts program, metadata about the production crew may apply to all tracks, metadata about a dancer may apply only to the picture track, and metadata about an orchestra may apply only to the sound track.

Both the DM segment and DM SourceClip have a TrackIDs property that references all the MXF tracks to which this metadata applies. If this property is omitted the metadata applies to all the tracks in the package.

8.8.4 How much metadata?

The theoretical limit to the number of metadata tracks in a package is huge (probably more than 2^{32}). Practical limits to do with the size of the header metadata and usefulness of such large numbers of tracks will have more of an impact on the actual upper limit.

8.8.5 Descriptive metadata identification

The final part of the plug-in mechanism worth mentioning here is identification of the scheme. SMPTE 377M defines generic Universal labels for the identification of MXF metadata plug-ins that are in a file and also defines generic keys for the wrapping of the descriptive metadata content.

Annex A (Informative)
Relationship of MXF to AAF

MXF was designed to have little or no divergence from the underlying AAF model. A joint working group ensured that any deviation of the two formats was justified. Both formats have benefited from the work carried out on the two different applications of the common underlying class model.

A.1 Comparison of MXF files with AAF files

For MXF files, the partitioning is designed for the following desirable characteristics:

1. Repetition of header metadata.
2. Incremental sequential writing.
3. The contents of the index tables do not change if the Index is relocated within the file.
4. The contents of the metadata KLV triplets do not change for each repetition.
5. The essence is completely unaffected by the insertion or deletion of partition, metadata or index sectors.
6. Multiple independent essence containers and index tables.
7. Simplicity – i.e., the ability for hardware only implementations to process MXF files.
8. The encoding of partitions does not require look ahead, except to record the number of bytes allocated for metadata and index.
9. The encoding of metadata and index segments does not require any knowledge of context within the multiplex.
10. The stream can be picked up safely (after failure or join in progress) at ANY body partition.
11. Minimal overhead.

For AAF files using structured storage, the overhead of this low-level data structure serves other needs, which are in conflict with some of the MXF requirements:

1. Efficient edit-in-place of individual metadata items and sets.
2. Complex hierarchical relationships between sets.
3. Efficient mixture of small and large data items.

To maximize interoperability, the MXF and AAF low-level byte-stream formats share some key concepts:

1. A common class model.
2. The widespread use of 16-byte Universal labels.
3. The notion of streams.
4. Stream identifiers (SIDs).
5. Essence descriptors.

Using this commonality, the AAF SDK can efficiently implement MXF, with the following desired characteristics:

1. An MXF device can always read and / or write MXF files.
2. An AAF application can always open MXF files with no conversion at all.
3. If an AAF application modifies an EDL in an MXF file, an MXF device would see this as updated header metadata.
4. If an AAF application adds AAF specific metadata to an MXF file, an MXF device would see the additions as "dark metadata".
5. An AAF application can flatten its internal object hierarchy to create an MXF file.
6. An AAF application can also, of course, create AAF files.
7. An AAF application can convert an AAF file into an MXF file in one of three ways (although not all of these methods will be built into the open source AAF SDK):
    a) filter out non-MXF data;
    b) constrain the creation of the AAF file so it is never beyond MXF complexity;
    c) render an AAF composition.

Note that metadata to describe effects other than cuts is defined by the advanced authoring format. Application-specific variants of MXF files including effects metadata could be defined. However, that is outside the scope of the MXF standard.

A.2 Relationship between MXF and AAF

Files created according to the MXF standard may be opened by applications that are designed to read AAF, and can be opened simultaneously by hardware or software designed for MXF.

Specific requirements on the MXF file format include:

Must permit precise and repeatable external references to defined positions in the contents of the MXF file.
Files created according to the MXF standard may be opened by applications that are designed to read AAF, and can be opened simultaneously by hardware or software designed for MXF.

A.3 MXF and AAF references – Inter-working

A.3.1 Strong references – In-file instance ownership

Strong references are about object instance ownership. Every object in AAF (and MXF) other than the ultimate root object is owned i.e. has one and only one strong reference to it. As a consequence, a valid MXF or AAF data model can be viewed as a tree-like hierarchy of strong references between objects.

An AAF data model persisted to Microsoft Structured Storage (MSS) represents a strong reference as MSS storage containment of the target object. This means strong references are efficiently followed in an AAF file.

An MXF data model represented in KLV uses unique object instance identifiers to identify the target of a strong reference. The target is persisted elsewhere in the KLV stream with the same identifier. (This is a bit like adding an “artificial” key column to a relational database table when no combination of the existing columns is unique for every possible row in the table.) These identifiers are transitory, in that provided this referential integrity is maintained, they may be re-generated whenever an object is persisted to file.

There is ongoing work to create a common representation in XML of the shared data model. Details of this work are available, at the time of writing, to the working committees of SMPTE, AAF and Pro-MPEG.

A.3.2 General weak references – In-file shared instances

The concept of general weak references exists in the design of both MXF and AAF. A general weak reference is a non-ownership reference to an object. Any object may be the target of zero or more weak references. Weak references allow information to be shared rather than duplicated.

The AAF data model never required general weak references and they are not currently implemented. A future implementation would probably use a path identifier (like a file system path) that identified the unique route from the root object to the referenced object via the strong reference tree.

The MXF data model uses general weak references in the DMS 1. General weak references are easily represented in MXF files by using the existing file-unique object instance identifiers.

A.3.3 Restricted weak references – Inter-file shared

The AAF data model does include shared objects. However these are all definitions of various kinds: data definitions, operation definitions, codec definitions, etc.

There are two common features of these classes:

1. The most important feature of each definition is that it has a universally unique identifier, which is the same across all files. These identifiers are defined by an application or by an external registry.

2. In the AAF data model, all object instances of each of these classes always reside in a known location in the strong reference hierarchy.

Each set of objects in a file (data definitions, codec definitions, etc.) is in effect a local copy of a universal registry, or at least all those entries that are used by the file. Each object includes the name of the definition and a short description, much the same as is found in a SMPTE registry.

Definitions are also represented using the universally unique identifier in MXF files. The only difference is that there is no local copy of the registry.

A.3.4 AAF Classes

The AAF data model is well documented and implementers involved in AAF – MXF interoperability are strongly advised to compare the latest version of the AAF class model and its implementation in the reference SDK against the latest version of SMPTE 377M.

Annex B (informative)
Preferred enumerated string values

This annex defines preferred string values for certain properties defined in SMPTE 377M using English terms and words.

Strings are listed in the form [SetName : PropertyName] in order to ensure that the target property is clearly identified.

B.1 Identification: Platform (operating system used)

The following values are preferred text string values that enable operating system type discovery for common software platforms. Other string values may be used where needed.

“Windows 95”
“Windows 98”
“Windows ME”
“Windows 2000”
“Windows NT”
“Windows XP”
“Mac OS Classic”
“Mac OS X”
“Unix System V”
“Solaris”
“Unix BSD 4.3”
“Unix BSD 4.4”
“Irix”
“Linux”
“FreeBSD”
“AIX”

Annex C (informative)
Bibliography

List of references used normatively in other parts of SMPTE 377M.

The following documents are referred to normatively in other parts of SMPTE 377M. The list is provided here for information so that a complete list of references can be found in a single place. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. For undated references, the latest edition of the normative documents referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

NOTES

1 This list may not include normative references for MXF documents defined after the publication point of this document (for example, new essence container documents).

2 Approved SMPTE standards may be obtained from http://www.smpte.org. Drafts of SMPTE documents may be obtained from ftp://smpte.vwh.net/pub. Approved ANSI standards may be obtained from http://www.ansi.org.

ANSI/SMPTE 298M-1997, Television — Universal Labels for Unique Identification of Digital Data

SMPTE 305.2M-2000, Television — Serial Data Transport Interface (SDTI)

SMPTE 314M-1997, Television — Data Structure for DV-Based Audio, Data and Compressed Video — 25 and 50 Mb/s

SMPTE 321M-2002, Television — Data Stream Format for the Exchange of DV-Based Audio, Data and Compressed Video over a Serial Data Transport Interface

SMPTE 322M-1999, Television — Format for Transmission of DV Compressed Video Audio and Data over a Serial Data Transport Interface

SMPTE 326M-2000, Television — SDTI Content Package Format (SDTI-CP)

SMPTE 330M-2004, Television — Unique Material Identifier (UMID)

SMPTE 331M-2004, Television — Element and Metadata Definitions for SDTI-CP

SMPTE 336M-2001, Television — Data Coding Protocol using Key-Length-Value

SMPTE 337M-2000, Television — Format for Non-PCM Audio and Data in an AES3 Serial Digital Audio Interface

SMPTE 338M-2000, Television — Format for Non-PCM Audio and Data in AES3 – Data Types

SMPTE 339M-2000, Television — Format for Non-PCM Audio and Data in an AES3 – Generic Data Types

SMPTE 352M-2002, Television (Dynamic) — Video Payload Identification for Digital Interfaces

SMPTE 356M-2001, Television — Type D-10 Type Stream Specifications — MPEG-2 4:2:2P@ML for 525/60 and 625/50

SMPTE 359M-2001, Television — Proposed SMPTE Standard for Television and Motion Pictures — Dynamic Documents

SMPTE 367M-2002, Television — Type D-11 Picture Compression and Data Stream Format

SMPTE 369M-2002, Television — Type D-11 Data Stream and AES3 Data Mapping over SDTI

SMPTE 370M-2002, Television — Data Structure for DV Based Audio, Data and Compressed Video at 100 Mb/s 1080/60i, 1080/50i, 720/60p

SMPTE 377M-2004, Television — Material Exchange Format (MXF) — File Format Specification

SMPTE 378M-2004, Television — Material Exchange Format (MXF) — Operational Pattern 1a (Single Item, Single Package)

SMPTE 379M-2004, Television — Material Exchange Format (MXF) — Generic Container

SMPTE 380M-2004, Television — Material Exchange Format (MXF) — Descriptive Metadata Scheme 1

SMPTE 381M, Television — Material Exchange Format (MXF) — Mapping MPEG streams into the MXF Generic Container

SMPTE 382M, Television — Material Exchange Format (MXF) — Mapping AES3 and Broadcast Wave Audio into the MXF Generic Container

SMPTE 383M-2004, Television — Material Exchange Format (MXF) — Mapping DV-DIF Data to the MXF Generic Container

SMPTE 384M, Television — Material Exchange Format (MXF) — Mapping of Uncompressed Pictures into the Generic Container

SMPTE 385M-2004, Television — Material Exchange Format (MXF) — Mapping SDTI-CP Essence and Metadata into the MXF Generic Container

SMPTE 386M-2004, Television — Material Exchange Format (MXF) — Mapping Type D-10 Essence Data to the MXF Generic Container

SMPTE 387M-2004, Television — Material Exchange Format (MXF) — Mapping Type D-11 Essence Data to the MXF Generic Container

SMPTE RP 204-2000, SDTI-CP MPEG Decoder Templates

SMPTE RP 210, Metadata Dictionary Registry of Metadata Element Descriptions

SMPTE EG 42-2004, Television — Material Exchange Format (MXF) — MXF Descriptive Metadata Engineering Guideline

AES3-2003, Serial Transmission Format for Two-Channel Linearly Represented Digital Audio Data

Draft AES Project X66 (Tentative Designation AES31-2): File Format for Transferring Digital Audio Data

EBU Tech T3285 Supplement 3 (2001): BWF, Peak Envelope Chunk

IEC 61834-2 (1998-08), Recording — Helical-scan Digital Video Cassette Recording System using 6.35mm Magnetic Tape for Consumer Use (525-60, 625-50 and 1250-50 Systems), Part 2: SD Format for 525-60 and 625-50 Systems

ISO/IEC 646:1991, Information Technology — ISO 7-Bit Coded Character Set for Information Interchange

ISO 13818-1:2000, Information Technology — Generic Coding of Moving Pictures and Associated Audio Information: Systems, 1996, (MPEG-2)

ISO/IEC 13818-2:2000, Information Technology — Generic Coding of Moving Pictures and Associated Audio Information: Video, 1996, (MPEG-2)

ISO/IEC 13818-2: Amendment 2: (MPEG-2, 4:2:2P@ML)

ISO/IEC 8825-1:1998, Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER) Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)

ITU-R BR.1352-1:2002, Broadcast Wave Format (BWF), Annex 1, Annex 1 Appendix 1 and 2, and Annex 3

C.1 Informative reading

The following list of informative documents is provided to help give background information and an overview of standards related to SMPTE 377M.

EBU / SMPTE Task Force for Harmonized Standards for the Exchange of Program Material as Bit-streams – 1998, http://www.smpte.org and http://www.ebu.ch

Advanced Authoring Format, http://www.AAFassociation.org

DVCPRO White Papers, http://www.dvcpropartners.com

The SMPTE Data Coding Protocol and Dictionaries, Jim Wilkinson, SMPTE Journal, July 2000 Vol. 109, No 7, Engineering Report

UNICODE – http://www.unicode.org for informative reading on the coding of international characters.
Pro-MPEG forum web site http://www.pro-mpeg.org

UML information for understanding class diagrams and other aspects of data modeling and programming http://www.oreilly.com