**SMPTE ST 2034-1:2017**
**Revision of**
**SMPTE ST 2034-1:2014**

# SMPTE STANDARD

# Archive eXchange Format (AXF) — Part 1: Structure & Semantics



## Table of Contents





Approved
February 27, 2017

## Foreword

SMPTE (the Society of Motion Picture and Television Engineers) is an internationally-recognized standards developing organization. Headquartered and incorporated in the United States of America, SMPTE has members in over 80 countries on six continents. SMPTE's Engineering Documents, including Standards, Recommended Practices, and Engineering Guidelines, are prepared by SMPTE's Technology Committees. Participation in these Committees is open to all with a bona fide interest in their work. SMPTE cooperates closely with other standards-developing organizations, including ISO, IEC and ITU.

SMPTE Engineering Documents are drafted in accordance with the rules given in its Standards Operations Manual.

SMPTE ST 2034-1 was prepared by Technology Committee 31FS on File Formats and Systems.

## Intellectual Property

At the time of publication, no notice had been received by SMPTE claiming patent rights essential to the implementation of this Engineering Document. However, attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. SMPTE shall not be held responsible for identifying any or all such patent rights.

## Introduction

This section is entirely informative and does not form an integral part of this Engineering Document.

The Archive eXchange Format (AXF) is an open format that supports interoperability among disparate data storage systems and ensures long–term availability of data, no matter how storage or file system technologies evolve. AXF inherently supports interoperability between existing, discrete storage systems, irrespective of the operating and file systems used, and also future–proofs digital storage by abstracting the underlying technology so that content remains available across generations of technology development.

At the most basic level, AXF is a file container that can encapsulate any number, size, and type of files in a fully self–contained and self–describing package. The package contains its own light-weight file system, which establishes independence from underlying operating systems, storage technologies, and file systems and can store any type of data on any type of storage media. Inside its packaging, AXF can contain metadata of any format, applicable to either AXF Objects or to individual files contained within AXF Objects; AXF also carries key preservation information, such as provenance, fixity, and the like — all key to ensuring long-term robustness and recoverability.

Historically, digital archive systems have used media data storage formats that are proprietary to their manufacturers, either intentionally or due to the lack of established standards. There have been neither interchange of media nor interoperability of archive systems between manufacturers and in some cases between different archive systems from the same manufacturer. Archives could be orphaned due to support ending for the systems used to create data archives. End users and manufacturers recognized that the proprietary nature of archive systems and the data stores that they create result in significant costs of operation that are unnecessary. These costs could be avoided if there were standardization of the format used for storage of the data on media and for transfer of the data between systems and locations. AXF permits separating the stored content from the systems that create and recover sets of data, thereby enabling refreshing of storage technology, recovering sets of data that otherwise would have been orphaned, and transferring sets of data between systems and locations.

This standard specifies a structure for data that can be written to any current or future data storage subsystem, regardless of the type of media on which it is stored. The data can include any types of files and associated metadata that are stored and transferred together in a structure called an "AXF Object." A single AXF Object can be spanned across multiple physical media, can be copied from one set of physical media to

another, and is agnostic to the Storage Media Type on which it is stored, e.g., spinning disc or linear tape. Regardless of the Storage Media Types on which they are stored, AXF Objects are identically structured and formatted for any given set and relationship of contained files and metadata.

AXF initially arose from the storage needs of the audiovisual production and archiving communities but quickly encompassed any type of file-based data. The transition to file-based workflows led to a new set of requirements throughout pre-production, production, distribution, storage, and preservation processes. Those requirements included long-term archiving of finished and unfinished materials, writing data to any type of storage subsystem using a standard scheme, transporting formatted archives between systems and locations using either media or networks, and allowing extensibility sufficient to accommodate any type of file, of any size, from any source, as well as adoption of any future storage technologies. AXF was created to address these requirements.

Audiovisual content archiving spans a wide range of content and data archiving systems and practices. At the time this standard was written, many different methods and media were commonly used to store file-based audiovisual content and its supporting information. Examples range from individual hard drives, solid state drives, and linear magnetic tape drives in small organizations to large spinning disc arrays in combination with very large robotic systems with multiple robots, each having multiple drives, in very large cultural, scientific, and legal archives. Applications in other industries that could benefit from the methods defined herein include medical imaging, geophysical exploration, scientific research, and similar high-volume producers of data.

The cultural, scientific, and business value of assets stored on these data systems is significant. Methods for storage, interchange, transport, and preservation of such assets, both locally and remotely, over both short and very long retention periods, demands a standardized, well-documented, non-manufacturer-specific method of writing data to any data storage system, from which the data then can be recovered and its contents used, updated, or transferred to another data storage system. All that would be necessary to achieve these objectives is a mechanism for recovering data from the media on which it is stored, plus utilities or applications that implement AXF.

The AXF standard creates a common method of writing individual files or related sets of files, and relevant metadata, onto data storage subsystems so that the structure of an AXF Object will remain the same no matter what vendor equipment or Storage Media Type is used. As long as the media remains viable and data can be read from that media, it will be possible to recover an AXF Object and unwrap its contents with a suitable utility or application running on whatever platform is current at the time. The AXF Object also has to be able to be recovered and stored on future data storage systems without requiring any changes to its contents simply to accomplish the act of medium migration, but it also needs to allow changes to its contents, in case updating is needed to data that already has been archived.

AXF addresses these needs through a combination of predefined eXtensible Markup Language (XML) schema fields, defined binary data structures that enable an AXF Object to carry any type of file within its File Payload, internal file system functionality, and key metadata enabling the spanning of AXF Objects across multiple physical media. The XML schema also enables essential information about an AXF Object and its contents to be read without having to process all the information within the AXF Object.

In addition to media interchange, AXF enables the interoperability of disparate systems through networks because it is structured as a streaming data set. Such interconnections enable seamless movement of AXF Objects from systems that create them, to systems that do not recognize the AXF protocol but store the AXF Object files nonetheless (perhaps in “cloud” storage), then to systems that are designed to recover data from AXF Objects.

Functionally, AXF acts like a file wrapper or a repository for all types of data without constraint. Unlike media-centric file formats such as MXF, which are similar in that they wrap essences, AXF can contain any number or types of files of any size encapsulated in an AXF Object. It is applicable across a much broader variety of file storage user groups than any media-specific file wrapper. Types of data can include media essence files, related metadata files, production files (such as word processing documents, hypertext documents, associated essence, applications, spreadsheets, and database copies), or any other type of data that users wish to store together. Unlike other file wrapper definitions, it is payload agnostic and does not require any special mappings or adaptations to accept the data an AXF Object carries.

AXF accommodates very large file sizes and quantities within AXF Objects. In the current version of this standard, 64-bit numbers are used to define the sizes of various parameters applicable to elements of AXF Objects. 64-bit numbers can express values up to 18.44674 x $10^{18}$ (e.g., 18.44674 petabytes). Use of 64-bit numbers thus can define file sizes in bytes, numbers of files, numbers of media in a spanned set, and similar characteristics up to 18.44674 x $10^{18}$ of any particular element. If future requirements exceed the number spaces provided in this document, there is nothing fundamental that limits any particular parameter to expression using a 64-bit number. Future revisions of this standard could adopt larger number spaces (e.g., 96-bit, 128-bit, etc.) for those parameters requiring them. The net result is effectively unlimited storage capability within AXF Objects, in terms of file sizes, numbers of files in an AXF Object, number of AXF Objects on a medium, number of media in a spanned set, and the like.

AXF enables updating AXF Objects when additions of, modifications to, or deletions of files or information that they contain are needed. The functionality to modify AXF Objects is provided by linking "Supplemental" AXF Objects, written into an archive system at a later time to an original ("Anchor") AXF Object. A Supplemental AXF Object updates contents of previous AXF Objects without requiring the original AXF Object itself to be modified. Since the original content of the AXF Object is retained in its original form, it is possible to restore either the original or the modified version whenever necessary. Additional Supplemental AXF Objects can be added in a chain, with restoration of the current or any earlier version possible at any time. When AXF Objects are refreshed by copying them to new media, it is possible to consolidate an Anchor Object and its Supplemental Object(s) into a single, new AXF Object. In doing so, it is possible to retain all of the constituent Objects of the Collected Set to which they belong, so that all earlier versions still can be reconstituted in the future.

AXF abstracts the storage of data from the applications that create AXF Objects and from the operating systems, file systems, drivers, and drives that store data on media. By this mechanism, any of the surrounding hardware and software components of systems can be replaced without affecting the data and its formatting within AXF Objects. A simplified view of where AXF fits into a basic stack is shown in Figure 1.

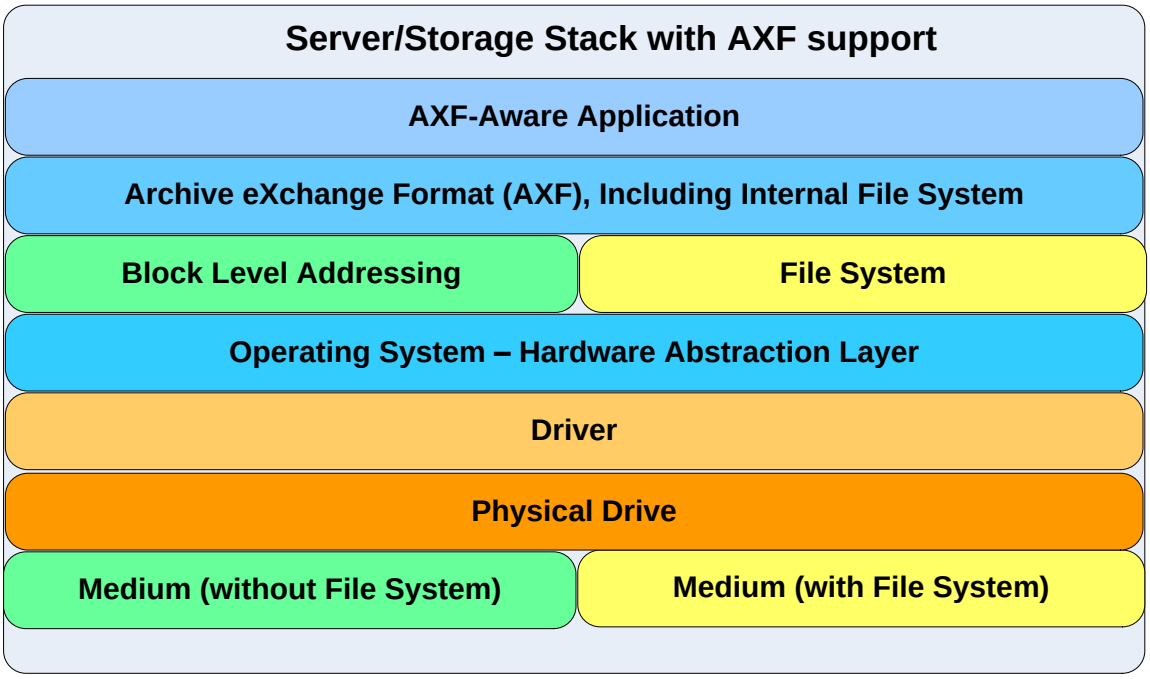



**Figure 1 – Hardware/Software Stack Incorporating AXF Writing to and Reading from Media**

AXF is designed so that each AXF Object comprises four main components, regardless of the technology that is used to store it. These components are:

Object Header – Each AXF Object begins with an Object Header, which contains descriptive XML metadata such as a unique identifier (UUID) for the AXF Object, information regarding its origin, its creation date, and a full index of all the files and folders contained in the Object, including file permissions and the like.

Generic Metadata Containers – Following an Object Header can be any number of optional Generic Metadata Containers. Such containers are self–contained, open metadata containers in which applications can place AXF-Object-specific metadata that is not part of the AXF Object File Payload. The metadata can be structured or unstructured, open or vendor-specific, binary, XML, plain text, or any other format.

AXF Object File Payload – Following any Generic Metadata Containers is the AXF Object File Payload. It contains the files encapsulated in the AXF Object. The File Payload consists of any number of triplets: File Data + File Padding + File Footer. File Padding ensures alignment of all AXF Object elements on the boundaries of Chunks into which each AXF Object is divided, thereby enabling addressing, by location within the AXF Object, by its internal file system. File Footer structures contain full information about the preceding file, along with a file-level checksum designed to be processed on-the-fly, with little or no overhead, during restore operations by an application. The information in File Footers enhances the resilience of AXF, as it can be used to recover File Payload data even if Object Header and Footer structures are missing or corrupt.

Object Footer – Completing an AXF Object is an Object Footer. It repeats the information contained in the Object Header and adds information captured during creation of the AXF Object, including per-file checksums, precise file sizes, and file positions within the AXF Object. The Object Footer is important to the interchange of an AXF Object because it allows efficient indexing by foreign systems when the media content is not previously known, thereby enabling media transport between systems that follow the AXF standard. It is one of the key structures that support the self-describing nature of AXF.

Other significant structures in the AXF protocol are AXF Medium Identifiers and AXF Object Indices. AXF Medium Identifiers are used on media to indicate formatting of the media according to the AXF protocol and to provide unique identification of the media. AXF Object Indices are optional compilations of the information in all Object Footers preceding each AXF Object Index on a medium, providing a single structure from which it is possible to obtain complete information on the contents of the preceding portion of a medium. When an AXF Object Index is the last structure on a medium, complete information about all AXF Objects stored on the medium can be obtained efficiently in one place.

AXF does not require a system to be fully compliant with this standard for it to be able to use and store AXF-generated AXF Objects. The initial adoption of AXF is anticipated to be in applications that create AXF Objects that then are stored on non-AXF-aware storage systems. Because the AXF Objects do not require a storage system to know that the AXF Objects are AXF-formatted, the AXF Objects will be viewed simply as files to be stored and retrieved. All that will be necessary to read AXF Objects will be the software and hardware needed to read the physical storage medium. As adoption grows, files can be moved into and out of AXF-aware systems as necessary, with the full range of features becoming available on systems that are AXF compliant. AXF-compliant applications will be able to read stored AXF Objects from any current operating system without unpacking entire AXF Objects to see critical metadata. Moreover, Archives, or AXF Objects within archives, also can span different types of media, allowing for flexibility within mixed-media archives and for AXF Objects to be identical, regardless of the media on which they are stored.

AXF offers resilience to data corruption and loss. AXF Object Indices, repeated identifier instances, and cryptographic hash checksums on both contents and AXF Objects allow for data corruption to be identified and mitigated. Even in catastrophic events, such as the loss of an external database containing records of the contents of an archive, the content database can be recreated by reading the archive and regenerating the archive-wide database from the records within the AXF Objects. This standard also enables the addition of more powerful data corruption recovery methods in future revisions by including provisions for incorporating forward error correction codes. AXF provides key features to identify and treat data corruption and loss across all types of storage formats. It allows big data and small to be processed in a consistent and standardized way.

# 1 Scope

This standard is Part 1 of a series of documents that specify a general-purpose format for the storage and/or communication of information in bulk form. The format is named the Archive eXchange Format (AXF). The format described is intended both for interchange between systems and to serve as a native format within systems.

This standard identifies two major categories of data storage media and specifies the basic structures of data stored on those Storage Media Types. It specifies a number of structural elements for use in constructing the appropriate structures for use on each of the Storage Media Types. It defines the semantics of data contained within fields specified for use in the structural elements. The structural elements themselves are documents coded in the eXtensible Markup Language (XML), and this document defines an XML Schema Description (XSD) file for use in formulating the XML documents to be used for the structural elements of AXF Objects.

# 2 Conformance Notation

Normative text is text that describes elements of the design that are indispensable or text that contains the conformance language keywords: "shall," "should," or "may." Informative text is text that is potentially helpful to the user, but not indispensable, and that can be removed, changed, or added editorially without affecting interoperability. Informative text does not contain any conformance keywords.

All text in this document is, by default, normative, except: the Introduction, any section explicitly labeled as "Informative," or individual paragraphs that start with "Note:".

The keywords "shall" and "shall not" indicate requirements strictly to be followed in order to conform to the document and from which no deviation is permitted.

The keywords, "should" and "should not" indicate that, among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited.

The keywords "may" and "need not" indicate courses of action permissible within the limits of the document.

The keyword “reserved” indicates a provision that is not defined at this time, shall not be used, and may be defined in the future. The keyword “forbidden” indicates “reserved” and, in addition, indicates that the provision never will be defined in the future.

A conformant implementation according to this document is one that includes all mandatory provisions ("shall") and, if implemented, all recommended provisions ("should") as described. A conformant implementation need not implement optional provisions ("may") and need not implement them as described.

Unless otherwise specified, the order of precedence of the types of normative information in this document shall be as follows: Normative prose shall be the authoritative definition; tables shall be next, followed by formal languages, then figures, and then any other language forms.

# 3 Normative References

The following standards contain provisions that, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

XML 1.0, World Wide Web Consortium (W3C) (2008, November) Extensible Markup Language 1.0 (Fifth Edition)

XML Namespaces 1.0, World Wide Web Consortium (W3C) (8 December 2009) Namespaces in XML 1.0 (Third Edition)

XML Schema Part 1 — World Wide Web Consortium (W3C) (28 October 2004), XML Schema Part 1: Structures (Second Edition)

XML Schema Part 2 — World Wide Web Consortium (W3C) (28 October 2004), XML Schema Part 2: Datatypes (Second Edition)

ISO/IEC 10646:2012, Information Technology — Universal Coded Character Set (UCS), Third Edition (1 June 2012)

ISO/IEC 19505-1:2012, Object Management Group (OMG) (August 2011), Unified Modeling Language: Infrastructure, v2.4.1

ISO/IEC 19505-2:2012, Object Management Group (OMG) (August 2011), Unified Modeling Language: Superstructure, v2.4.1

IETF RFC 1494, Equivalence Between 1988 X.400 and RFC-822 Message Bodies (August 1993)

IETF RFC 2231, MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations. (November 1997)

IETF RFC 3629, UTF-8, A Transformation Format of ISO 10646 (November 2003)

IETF RFC 3986, Uniform Resource Identifier (URI): Generic Syntax

IETF RFC 4122, A Universally Unique IDentifier (UUID) URN Namespace. (July 2005)

IETF RFC 6657, Update to MIME Regarding “Charset” Parameter Handling in Textual Media Types (July 2012)

IETF RFC 6838, Media Type Specifications and Registration Procedures

ISO/IEC 1001:2012(E) (VOL1), Information Technology — File Structure and Labeling of Magnetic Tapes for Information Interchange (2012-08-01)

ISO/IEC 13239:2012 (CRC), Information Technology — Telecommunications and Information Exchange Between Systems -- High-Level Data Link Control (HDLC) Procedures

FIPS PUB 180-4, Federal Information Processing Standards Publication, Secure Hash Standard *SHS), Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, March 2012

UTC, Coordinated Universal Time, ITU-R Recommendation TF 460-6, Standard-Frequency and Time-Signal Emissions, International Telecommunications Union, Radio Sector, Geneva, Switzerland, February 2002

# 4 Definitions

The following terms shall have the definitions specified for them below, both throughout this document and in related Parts of this SMPTE standard.

### 4.1 Anchor Object

The first AXF Object in a Collected Set of AXF Objects, the **SetSequence** value of which is 1. It is the AXF Object from which all Subsequent Objects in the Collected Set differentiate the contents of the Product Object.

### 4.2 Archive Object

An AXF Object.

### 4.3 Attribute

An XML markup construct consisting of a name/value pair that describes an element.

### 4.4 AXF

Archive eXchange Format.

### 4.5 AXF-Aware

Capable of reading, interpreting, and/or writing AXF Objects and other AXF data structures.

### 4.6 AXF Media

Multiple media prepared to carry AXF Objects.

### 4.7 AXF Medium

A medium prepared to carry AXF Objects.

### 4.8 AXF Medium Identifier

A data structure at the start or root of a medium that indicates that it has been prepared for storage of AXF Objects and that provides information about the identification, formatting, and history of the medium.

### 4.9 AXF Object

A group of data elements that are stored as a unit and that are contained within a single data structure.

### 4.10 AXF Object Index

A data structure, stored on a medium, that is a collection of the complete information about all previously-stored AXF Objects on that medium and the contents of those AXF Objects, as well as the locations of the content files stored within those AXF Objects.

### 4.11 Base Name

The fundamental name of a file; the portion of a file name prior to any file name extension that identifies the type of file or structure.

### 4.12 Big Endian

The order in which the bytes of a multi-byte number are transmitted with the most significant byte first.

### 4.13 Binary Structure Container

A data structure that is formed using easily identifiable binary data patterns of UTF-8 characters and that carries a payload of a specified type for the purpose of providing structural elements for AXF Objects and for the files contained within them.

### 4.14 Binary Structure Container Payload

The data contained within the **Payload** field of a Binary Structure Container.

### 4.15 Block

A section of data of uniform size (measured in bytes) as stored on a physical medium that is independently addressable for write and read operations, and which is defined as a “block” on certain Storage Media Types and as a “sector,” a “page” or some other term on other Storage Media Types.

### 4.16 Block-Based

Media that enable direct access to Blocks of data stored thereon and that do not incorporate or depend upon File Systems.

### 4.17 Block Position

The location on a medium of a block of data, measured as the number of blocks from the beginning of the medium, as counted by the drive on which the medium is carried.

### 4.18 Block Size

A parameter that defines the size of Blocks contained on a medium. Some Storage Media Types can accommodate varying block sizes across the storage space of a medium.

### 4.19 Byte Order Mark

A small sequence of bytes carried within a file from which it is possible to determine the byte ordering (Big Endian or Little Endian) of the contents of the file. (Abbreviated BOM.)

### 4.20 Chunk

A section of data of uniform size (measured in bytes) within an AXF Object.

### 4.21 Chunk Size

A parameter that defines the size of all of the Chunks within an AXF Object.

### 4.22 Collected Set

A group of AXF Objects, consisting of an Anchor Object and one or more Subsequent Objects, that, when compiled, produce a Product Object differing from the Anchor Object as a result of additions, replacements, or deletions of files.

### 4.23 Content File

A File of any type that is stored as part of the File Payload within an AXF Object. Also a Payload File.

### 4.24 Data Element

A file or data structure that forms part of the overall data stored within an AXF Object or on a medium.

### 4.25 Element

An XML logical document component bounded by matching start and end tags and including any content between the tags plus any associated attributes.

### 4.26 Embedded File System

A light weight File System having its database contained within an AXF Object.

### 4.27 Endianness

The ordering of bytes within a multi-byte number. Byte orders typically are Little Endian or Big Endian, although other, less frequently used byte orders exist.

### 4.28 File Mark

A data structure written or read by a Linear Media drive to provide separation between data elements stored on a medium using the media technology for which the drive is designed.

### 4.29 File Footer

A Binary Structure Container that carries a payload with information to identify the file that it follows and to provide detailed information about that file.

### 4.30 File Padding

Data inserted at the end of a file to completely fill the last Chunk occupied by the file.

### 4.31 File Payload

The portion of an AXF Object that carries the Payload Files and their associated File Footer Binary Structure Containers.

### 4.32 File-System-Based

Media that incorporate and depend upon File Systems and provide no direct access to Blocks of data.

### 4.33 File System

A type of data store that can be used to store, retrieve, and update a set of files.

### 4.34 File Tree

A hierarchical structure of folders and files, expressed in XML, that starts at a root point and establishes relationships between folders and the folders and files that they contain.

### 4.35 Fixity

Fixity is the property of being constant, steady, and stable. Fixity checking is the process of verifying that a digital object has not been altered or corrupted. In practice, this is most often accomplished by computing and comparing cryptographic hash checksums.

### 4.36 Fragment

A data structure that contains a portion of an AXF Object that is linked to preceding or following Fragments, or both, through shared UUID values.

### 4.37 Generic Metadata Container

A Binary Structure Container the payload of which is metadata.

### 4.38 Linear Media

Media requiring writing and reading sequentially along the length of the medium (typically tape).

### 4.39 Little Endian

The order in which the bytes of a multi-byte number are transmitted with the least significant byte first. This might or might not match the order in which numbers are normally stored in memory for a particular processor.

### 4.40 Media

Plural of medium.

### 4.41 Media Class

The general category of Media, that is, Block-Based Media (i.e., both Linear and Non-Linear Media without File System support) and File-System-Based Media (i.e., both Linear and Non-Linear Media with File System support).

### 4.42 Media Family

A group of related Storage Media Types sharing a common set of media technology but allowing for different implementation characteristics, such as with respect to storage density.

### 4.43 Media Technology

A set of techniques used to store data on a particular form of media, including the choice of medium itself and the methods for such functions as recording bits on a medium, error correction, indicating a separation between portions of data stored, and the like.

### 4.44 Media Type

A registered name that describes the format of the content of a file.

### 4.45 Medium

A physical carrier on which data is stored, which comprises an integer number of Blocks and which can be either Block-Based or File-System-Based. (Equivalent to **Storage Medium**.)

### 4.46 Non-Linear Media

Media that can be written, read, and accessed in a non-sequential manner, enabling data to be accessed randomly on such a medium.

### 4.47 Object Footer

A data structure at the end of an AXF Object that provides complete information about the contents of the AXF Object, the positions within the AXF Object of the files that it contains, and other information about the AXF Object and its contents.

### 4.48 Object Fragment

The portion of an AXF Object stored on a single medium when the AXF Object is spanned across multiple media.

### 4.49 Object Header

A data structure at the start of an AXF Object that provides some or all of the information about the contents of the AXF Object, the positions within the AXF Object of the files that it contains, and other information related to the AXF Object and its contents.

### 4.50 Padding

Data having a value of 0x00 used to fill data space to maintain the format of a data structure.

### 4.51 Padding Chunk

A Chunk completely filled with bytes having a value of 0x00.

### 4.52 Payload File

A file contained within an AXF Object following the Payload Start Binary Structure Container and preceding the Payload Stop Binary Structure Container, if present.

### 4.53 Product Object

The compilation of files, directory structure, and other information that results from the combination of the Anchor Object and all subsequent AXF Objects in a Collected Set.

### 4.54 Spanned Set

A group of media containing Fragments of a single AXF Object linked together by a shared UUID identifying the AXF Object and by shared UUID values matching the Span Points of the AXF Object.

### 4.55 Spanning

A process for fragmenting a single AXF Object and storing its fragments on multiple media.

### 4.56 Span Point

The location within an AXF Object at which the AXF Object contents are fragmented for storage in separate Fragments.

### 4.57 Storage Media Type

A specific combination of media technology with parameters for the various data storage characteristics and data processing methods.

### 4.58 Storage Medium

A Medium used to store data. (Equivalent to **Medium**.)

### 4.59 Subsequent Object

An AXF Object in a Collected Set of AXF Objects other than the Anchor Object, the **SetSequence** value of which is greater than 1. It is an AXF Object that differentiates the contents of the Product Object from the Anchor Object of the Collected Set by indicating additions, replacements, or deletions of files and that carries any files necessary to effectuate the additions or replacements of files that it indicates.

### 4.60 Symbolic Link

A pointer contained within a directory structure to a file or folder that may be internal or external to the file hierarchy described by the directory structure. Symbolic Links are used with certain types of File Systems, while other types of File Systems use “shortcut” files referenced by the directory structure to perform the pointer function.

### 4.61 Symlink

Symbolic Link

### 4.62 URI

Uniform Resource Identifier – A compact sequence of characters that identifies an abstract or physical resource.

### 4.63 UUID

Universally Unique Identifier – An identifier that is unique for all practical purposes and is created according to an algorithm specified to lead to different values for each such identifier generated.

### 4.64 VOL1

A label applied at the beginning of data tape media according to the provisions of ISO/IEC 1001 “Information technology – File structure and labeling of magnetic tapes for information interchange”

## 5 Storage Media Types

Media that carry AXF Objects in conformance with this standard fall into one of two fundamental categories: Block-Based and File-System-Based. The formatting of each of these Storage Media Types is according to specific arrangements described generally in this section and in greater detail below in other sections. As also described below, AXF Objects stored on the respective Storage Media Types do not differ in their basic formats, but certain indexes and pointers do differ somewhat in the values they carry depending upon the Storage Media Types upon which their AXF Objects are stored. Mechanisms are provided herein and in other Parts of this document suite for the storage, transfer and processing of AXF Objects between Storage Media Types in a transparent manner and using minimal processing. Block-Based and File-System-Based Media are either Linear or Non-Linear.

Linear Media are media that only can be written and read sequentially, from end to end. In the drives that write and read them, Linear Storage Media Types include mechanisms for writing and reading File Marks or similar identifiers of points of separation between sequential sections of data stored on the media. Linear Storage Media Types include in the drives that write and read them at least the capability to search for File Marks, to count File Marks over the length of a medium, and to advance in either direction by a specified number of File Marks. Typical Linear Storage Media Types are based upon the use of electromagnetic tape for the storage of data. Linear Media carry File Systems (e.g., LTFS) or do not carry File Systems.

Non-Linear Media do not use File Marks and instead are addressed using logical block addresses that point to specific locations where data are stored. Non-Linear Media provide the ability to move directly from any place on the medium to any other place on the medium. Typical Non-Linear Storage Media Types are based upon the use of magnetic or optical disks or flash memory devices for the storage of data. Non-Linear Media generally are expected to be File-System-Based, but the AXF protocol will support Block-Based variants as well.

Within AXF Objects, pointers are provided in several data structures to enable retrieval of files stored within the AXF Objects. The pointers indicate the locations of files within AXF Objects using either relative or absolute addresses based on Chunk counts within the AXF Objects. Relative addressing is used when pointing to all structures contained within an AXF Object. Absolute addressing is used when pointing to the starting location of an AXF Object. Combining relative and absolute addresses enables determining absolute locations of all structures within an AXF Object on the medium on which it is stored.

## 5.1 Media with File Systems

In the AXF context, File-System-Based Media shall be media that store the necessary indexes for the files that they contain and operate with file system software that manages those indexes and the data storage and recovery processes on the media. File systems provide services that include at least indexing mechanisms for files and their storage locations on media, management of the locations of files when writing to the media, and recovery of files with their data presented in the order in which they were stored.

Example: File Systems that can be used to store AXF Objects include the File Allocation Table (FAT), the New Technology File System (NTFS), Universal Disk Format (UDF), Distributed File System (DFS), Network File System (NFS), Hierarchical File System Plus (HFS+), Linear Tape File System (LTFS), and many others. On any File System, AXF Objects are stored as files, providing a constant AXF Object format, thus isolating the contents of the AXF Objects from the File Systems.

On File-System-Based Media, the locations of AXF Objects are managed by the File Systems; thus, it is unnecessary to include absolute addresses of the starting locations of AXF Objects within those AXF Objects. Consequently, in the storage of AXF Objects on Media having File Systems, the fields that otherwise would carry the absolute addresses of the AXF Object starting locations should be set to a value of -1. When transferring AXF Objects between Storage Media Types, the values in such fields normally will have to be modified to meet this requirement. Such transfers are discussed in more detail in Section 7.4.

## 5.2 Media without File Systems

Media without File Systems do not provide File-Based references to the content they carry and are referred to as Block-Based Media. They are based on storage of data in blocks of specific sizes, but many of these Storage Media Types lack means for determining the numbering of blocks on a medium. These media either can be Linear or Non-Linear in nature. On Linear Media, File Marks or similar mechanisms typically are used to enable systems writing to them to place specific reference points along media. Writing of File Marks can be a time-consuming process, however, on many Storage Media Types without File Systems. For this reason, the AXF protocol eliminates use of File Marks in all but two instances at the start of each linear AXF Medium.

The fundamental requirement for inclusion of File Systems on Linear Media is the ability of the media to be partitioned. Versions of Linear Storage Media Types that do have the capability for partitioning of the media nevertheless can be used without File Systems when applications can benefit from such utilization. Early

generations of many Linear tape families are incapable of being partitioned and consequently cannot support File Systems, while later generations of those families of Linear tape types are capable of such support.

On Block-Based AXF Media, the locations of AXF Objects are not managed by file systems. Consequently, AXF Objects include fields for the absolute addresses of Object Headers on such AXF Media. The absolute addressing used requires the system writing to or reading from a medium to keep track of the block count along the medium as it writes to it or reads from it. In the storage of AXF Objects on Block-Based Media, the fields carrying the absolute addresses of the AXF Object starting locations are set to the positions of the blocks containing the first bytes of the Object Header binary structure containers, the locations of which are being identified. When transferring AXF Objects between Storage Media Types, the values in the fields containing the Object Header absolute addresses should be modified to point to the correct locations. Such transfers are discussed in more detail in Section 7.4.

### 5.3 File Marks

File Marks are data patterns that can be placed on magnetic Linear Media (e.g., data tape) by drives, upon instructions to do so from controlling applications. The data patterns are predetermined for each particular type of medium and are intended to be easily recognized by drive subsystems when reading or scanning a medium at high speed.

The AXF protocol avoids the use of File Marks to the extent possible to improve the efficiency of tape operations. Only two are used: One following the VOL1 identifier and the other following the AXF Medium Identifier on a Linear Block-Based AXF Medium. See Section 5.4 for the designation of Storage Media Types on which File Marks shall be applied.

### 5.4 Relationships Between AXF Structures and Storage Media Types

Fundamentally the same AXF Object data structures are stored on all Storage Media Types, but certain media formatting and identification data structures shall be applied to particular Storage Media Types according to the relationships shown in Table 1. There are four categories of media to which the formatting or identification structures are applied: Block-Based and File-System-Based Linear Media and Block-Based and File-System-Based Non-Linear Media, as represented by the four columns in the right-hand portion of the table. The various structures listed in the leftmost column are described in the sections of this standard enumerated in the next-to-leftmost column of the table.

In Table 1, structures are indicated as being required, not applicable, or optional. Structures indicated as “Required” shall be employed on media in the specified media categories, except that File Marks shall be required only in instances in which the hardware media drives support them. Structures indicated as “Not Applicable” shall not be employed on media in the specified media categories. Structures indicated as “Optional” may be employed on media in the specified media categories.

Note: The VOL1 and File Mark structures are shown in Table 1 as “Not Applicable” in certain instances. They are not applied according to the AXF protocol, but they nevertheless can be present on a medium carrying AXF structures if applied by an underlying file system or other media management protocol.

**Table 1 — Relationships between AXF Structures and Storage Media Types**

| AXF Structure | | Linear Media | | Non-Linear Media | |
|---|---|---|---|---|---|
| | Section | Block-Based | File-System-Based | Block-Based | File-System-Based |
| **VOL1 Label** | 6.4.2.1 | √+ | – | √ | – |
| **File Mark** | 5.3 | √* | – | – | – |
| **AXF Medium Identifier** | 0 | √+ | O | √ | O |
| **AXF Object** | 6.4.3.1 | √ | √ | √ | √ |
| **AXF Object Index** | 6.4.2.3 | O | O | O | O |
| **Example** | | Block-Based Data Tape | LTFS-Formatted Data Tape | Block-Based Disk or Flash Drive | NTFS Disk or FAT32 Flash Drive |

**√ = Required, – = Not Applicable, O = Optional, + = Followed by File Mark, * = When Supported**

# 6 Archive eXchange Format (AXF) Structure

The Archive eXchange Format (AXF) describes a protocol that includes an Archive Object (hereinafter "AXF Object") data structure in which related content and information about that content are encapsulated and stored when interchanged between storage systems and in which data can be stored internally within systems that make use of the protocol. AXF also describes a storage arrangement for AXF Objects and other data that is applied to media used for the carriage of AXF Objects. Associated with the AXF Objects and storage arrangement are a number of identifiers, metadata, and data structures that are used to construct specific instances of AXF Objects stored on AXF Media. Included among the identifiers, metadata, and data structures are those that permit the spanning of single AXF Objects across multiple AXF Media and others that allow for indexing of AXF Media by systems.

All of the AXF Object data structures and File Payload data are Chunk-aligned internally. This enhances resiliency and assists in the recoverability of packages in the event of corruption of the embedded indexing structures. The Chunk size may be one byte in length; however this practice normally should be avoided, as it increases index and recovery complexity. Nevertheless, it can be applied when it is necessary for systems to avoid certain restrictions of Chunk-aligned structures. As a result of the internal Chunk alignment and indexing structures, AXF Objects are said to include Embedded File Systems that assist in the abstraction of the underlying storage technology, medium, File System, and operating system.

AXF Objects are constructed so that they can be transferred sequentially when necessary (e.g., to a Linear medium or from an archive system to a remote storage sub-system over a data circuit or communications link). Sequential transfers allow AXF Objects to be streamed to media in write-once fashion, supporting, for example, linear and write-once media as well as processing during transfers across data circuits prior to completion of the transfers of complete AXF Objects.

## 6.1 Form of Data Expression

The Payloads carried within Binary Structure Containers (see Section 6.4.1.2) to form all of the AXF data structures contained on AXF Media and/or within AXF Objects formatted according to this standard shall be expressed in the form of documents in the eXtensible Markup Language (XML), as specified in XML 1.0 and XML Namespaces 1.0. XML documents created according to this standard shall be generated using the XML Schema Description (XSD) files described in Section 10. The XML Schema Description (XSD) method is specified in XML Schema Part 1, and XML Schema Part 2.

## 6.2 Byte Order

The byte order of the Binary Structure Container (see Section 6.4.1.2) other than the binary Structure Container Payload and those fields that are of the UTF-8 data type shall be little endian. The byte order of a Binary Structure Container Payload shall be indicated by the **Payload Format** field as specified in Table 2. The byte order of a Payload File (see Section 6.4.3.7) may be signaled in the associated File metadata (in the XML File data structure); it may be signaled by a Byte Order Mark contained within the Payload File; or it may be unknown. When a Byte Order Mark is known to be contained within a Payload File, that information also may be signaled in the associated File metadata. In cases in which the byte order or the presence of a Byte Order Mark is unknown, no entry with respect to the byte order shall be made in the File metadata.

## 6.3 General AXF Concepts

AXF deals with the limitations of legacy storage formats through application of multiple solutions, employed in parallel with one another: AXF employs a variable length FileTree structure that can list an unlimited number of files. The sizes of files stored within an AXF Object are expressed as XML integers, and XML integers have no fundamental limit in their size. AXF establishes hierarchical relationships between files and folders through inclusion of a FileTree structure and can include an unlimited number of entries on an unlimited number of levels within a hierarchy. AXF expresses file names as XML strings of unlimited size in UTF-8, and UTF-8 can express all Unicode characters in single bytes or multiple-byte groups. AXF provides a mechanism for exceeding the physical limits of the media on which AXF Objects are stored through provisions for Spanning to an unlimited number of media, thereby avoiding any potential limitation on the aggregated size of an AXF Object. AXF provides a mechanism for altering previously-stored AXF Objects, even on write-once media, through use of Collected Sets, which enable additions to, deletions from, and replacements of files in the File Payloads of AXF Objects, without the need to reconstruct or rewrite any previously written AXF Objects.

These concepts are discussed in general terms in this Section (6.3) and in detailed terms in Section 6.4.

### 6.3.1 Embedded File System / File Tree / File Payload

The combination of a File Tree structure and the Payload Files of an AXF Object form an Embedded File System, wholly contained within the AXF Object. AXF establishes relationships between the files in AXF Object File Payloads and folders in which they are contained, through a File Tree structure expressed in XML. Folders in AXF Objects exist solely as listed entries within the File Tree XML structure. The File Tree structure of an AXF Object lists all of the folders and the files within the folders, numbering them in such a way that their relationships can be determined by reading the FileTree data type. Multiple copies of the FileTree data type are stored within an AXF Object and, optionally, elsewhere on the medium on which the AXF Object is stored, to provide redundancy and to support resilience of Payload File recovery. Within an AXF Object, the FileTree data type is stored in the Object Header and Object Footer; external to the AXF Object, the FileTree data type can be stored in one or more Object Indices stored on the medium on which the AXF Object resides, as appropriate for the medium. In the File Payload, files are followed by File Footer XML structures that both demark the separation between one file and the next and provide all of the detailed information about the files that they follow.

### 6.3.2 Spanning

When an AXF Object contains too much content to fit in the remaining space on a medium, it can be divided into two or more fragments that can be stored on multiple media. The process is called Spanning and results in a Spanned Set of AXF Objects, each Object containing one fragment of the overall AXF Object. The Spanned Set is structured in such a way that the overall AXF Object can be reconstructed easily from its fragments. The reconstruction is aided by a series of linkages between the fragments using specialized headers and footers that carry identifiers for their respective fragments and the links between them.

### 6.3.3 Collected Sets

When it is desired to modify an AXF Object, through addition, substitution, or deletion of Payload Files, it is possible to do so through creation of a Subsequent Object that carries instructions on the modifications to be made and any additional Payload Files that can be necessary when carrying out the instructions. In such a case, the original AXF Object becomes known as an Anchor Object, and one or more Subsequent Objects can be added to what becomes a Collected Set of AXF Objects. Through compilation of the AXF Objects in a Collected Set in sequence, it is possible to produce a Product Object at any level of revision of the Collected Set that is needed. This mechanism permits modification of AXF Objects written on media that are not rewritable and also preserves all versions of the Collected Sets without loss of any content.

## 6.4 AXF Data Structures

Data structures generally utilized within the AXF protocol, and described in the following sections, are divided into these categories:

- General Data Structures (see Section 6.4.1)
    - Chunk (see Section 6.4.1.1)
    - Binary Structure Container (see Section 6.4.1.2)
- Media Data Structures (see Section 6.4.2)
    - VOL1 Label (see Section 6.4.2.1)
    - AXF Medium Identifier (see Section 6.4.2.2)
    - AXF Object Index (see Section 6.4.2.3)
- AXF Object Data Structures (see Section 6.4.3)
    - File Tree (see Section 6.4.3.3)
    - Object Header (see Section 6.4.3.4)
    - Object Footer (see Section 6.4.3.10)
    - Generic Metadata Container (see Section 6.4.3.5)
    - File Payload Start (see Section 6.4.3.6)
    - File Payload (see Section 6.4.3.7)
    - File Footer (see Section 6.4.3.8)
    - File Payload Stop (see Section 6.4.3.9)
    - Fragment Footer (see Section 6.4.3.11)
    - Fragment Header (see Section 6.4.3.12)

### 6.4.1 General Structures

General structures are part of the AXF protocol that transcend AXF Media- and AXF Object-specific structures, as defined in subsequent sections of this standard. These general structures have a wide applicability and are intended to ensure uniformity, regardless of the particulars of the specific implementation, Storage Media Type, or AXF Object to which they apply.

#### 6.4.1.1 Chunks

Requirements for the sizes and alignment of Chunks on media are different for Block-Based and File-Based systems.

##### 6.4.1.1.1 Block-Based Systems

In Block-Based systems, Chunk boundaries shall be aligned to Block boundaries. The first byte of an AXF Object shall align with the first byte of a Block. All AXF Objects on a given Medium shall have the same Chunk size, matching the Block size of the Medium.

Note: When reading or writing AXF Objects on Block-Based systems, chunk or block realignment only can be performed by AXF–aware applications.

##### 6.4.1.1.2 File-System-Based Systems

In File-System-Based systems, Chunk Boundaries need not be aligned with Media internal structures. Different AXF Objects on the same medium may have different Chunk sizes. Applications that read or write AXF Objects on File-System-Based systems need not be AXF aware with respect to AXF Chunk or Block realignment.

Note: On File-System-Based Media, it is possible for an AXF-aware application to tune Chunk sizes of AXF Objects for best efficiency based upon minimum, maximum, and average file sizes.

#### 6.4.1.2 Binary Structure Container

A Binary Structure Container is a mechanism, expressed in binary form, for encapsulating payloads for the multiple data structures that comprise the overall AXF protocol. It is used to create both AXF Media data structures and AXF Object data structures, providing a universal approach to creating, storing, interpreting, and parsing them. A Binary Structure Container can be seen as a well-defined and predictable encapsulation, or wrapper, for each structure (payload) that can be contained within it. Binary Structure Containers are applied to all AXF Media and AXF Object data structures except Payload File data itself.

Binary Structure Containers are used to avoid the complexities associated with raw storage of XML and other data types on various types of media and to ensure bit fixity and the ability to validate these structures during processing. Binary Structure Containers add resiliency to the overall AXF protocol, allowing applications to detect, and potentially recover, key structures independent of other index and metadata structures that are part of an AXF Object or AXF Medium. A Binary Structure Container is a generic container that can contain any type of data in its Payload. A Binary Structure Container also serves as the foundation for a Generic Metadata Container, as described in Section 6.4.3.5.

Binary Structure Containers shall be structured as defined in Table 2 and illustrated in Figure 2. Unless otherwise stated, all values in Table 2 shall be mandatory.

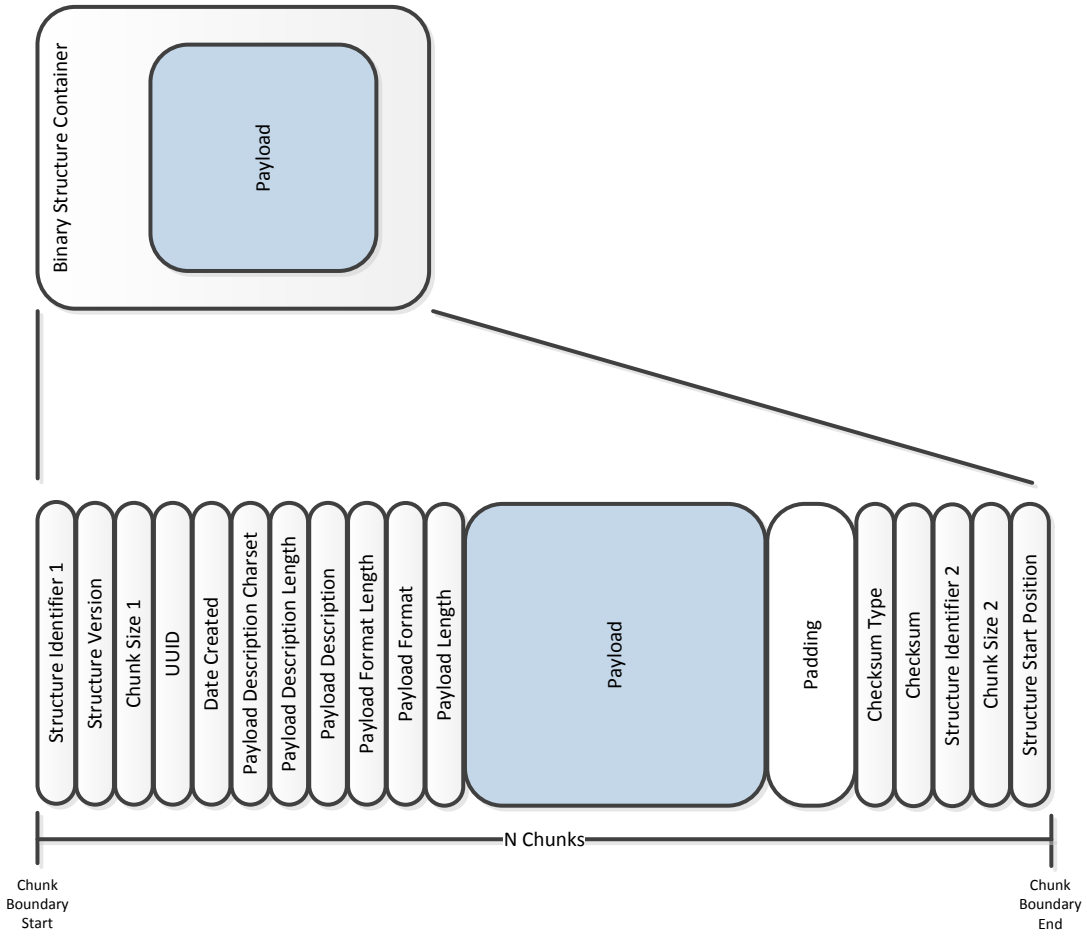



**Figure 2 – Binary Structure Container Overview**

## Table 2 – Structural Elements of Binary Structure Containers

| Field Name | Byte Offset | Length (Bytes) | Data Type | Example | Description and Requirements |
|---|---|---|---|---|---|
| Structure Identifier 1 | 0 | 32 | UTF-8 | AXF_OBJECT_HEADER | The **Structure Identifier 1** value shall be a string of UTF-8 characters identifying the type of data structure within the **Payload** of the structure. Specified character strings shall be used by applications to identify contained data structures. The string shall be padded with UTF-8 NUL characters to the field boundary. UTF-8 NUL characters shall not be used within structure identifier values. A duplicate of this information is contained in the **Structure Identifier 2** field.<br><br>The first bit of each **Structure Identifier 1** shall coincide with the first bit of the Chunk in which it is contained.<br><br>NOTE: The following values identify the AXF data structures defined in this standard:<br><br>AXF_OBJECT_HEADER<br>AXF_OBJECT_FOOTER<br>AXF_OBJECT_METADATA<br>AXF_OBJECT_FILE_PAYLOAD_START<br>AXF_OBJECT_FILE_PAYLOAD_STOP<br>AXF_OBJECT_INDEX<br>AXF_OBJECT_FRAGMENT_HEADER<br>AXF_OBJECT_FRAGMENT_FOOTER<br>AXF_MEDIUM_IDENTIFIER<br>AXF_FILE_FOOTER |
| Structure Version | 32 | 4 | Unsigned 32-bit integer | 1 | The **Structure Version** shall be the version identifier for the container. This particular version information is specific only to Binary Structure Containers. All Binary Structure Containers within an AXF Object shall be of the same **Structure Version**. Binary Structure Containers constructed according to this version of this standard shall have a **Structure Version** value of 1). |

| Field Name | Byte Offset | Length (Bytes) | Data Type | Example | Description and Requirements |
|---|---|---|---|---|---|
| Chunk Size 1 | 36 | 8 | Unsigned 64-bit integer | 100000h | The **Chunk Size 1** shall indicate in bytes the Chunk size used for the current AXF Object. The same value is duplicated in the **Chunk Size 2** field.<br><br>Example: A value of 100000h indicates all Chunks within the associated AXF Object have a size of 1,048,576 (decimal) bytes. |
| UUID | 44 | 16 | Unsigned 128-bit integer | 90997230098211E2892E0800200C9A66(h) | The **UUID** value shall be equal to the **AXF Object UUID**, as specified in the associated XML payload carried within the Binary Structure Container, when the Binary Structure Container applies to AXF Object-specific structures such as Object Header, Object Footer, File Footer, etc. All Binary Structure Containers associated with the same AXF Object shall contain the same **AXF Object UUID** value.<br><br>The **UUID** value shall be equal to the **AXF Medium UUID**, as specified in the associated XML payload carried within the Binary Structure Container, when the Binary Structure Container applies to AXF Media-specific structures, i.e., AXF Medium Identifiers and AXF Object Indices. All Binary Structure Containers associated with the same AXF Medium shall contain the same **AXF Medium UUID** value. |
| Date Created | 60 | 8 | Signed 64-bit integer | 1348846975 representing "Fri Sep 28 11:42:55 EDT 2012" | The **Date Created** value shall indicate the time at which the Binary Structure Container in which it is found was created. Its value shall be the elapsed time, since 00:00:00 UTC on January 1, 1970, expressed as a number of seconds. |
| Payload Description Encoding Form | 68 | 40 | UTF-8 | UTF-8 | The **Payload Description Encoding Form** field shall provide the designation of the encoding form used in the **Payload Description** field to encode the scalar values of the characters contained in the payload description. Allowed values shall be: UTF-8, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32LE, and UTF-32BE. The encoding form shall be as defined in ISO/IEC 10646, as shall be the scalar values associated with the characters and ideographs. |

| Field Name | Byte Offset | Length (Bytes) | Data Type | Example | Description and Requirements |
|---|---|---|---|---|---|
| Payload Description Length | 108 | 2 | Unsigned 16-bit integer | 128 | The **Payload Description Length** field shall indicate the length of the **Payload Description** field expressed in bytes. |
| Payload Description | 110 | Variable length, defined by the **Payload Description Length** field | Binary data encoded as specified in **Payload Description Encoding Form** | Migration Metadata, Proxy, Vendor Metadata, JPEG Cover Art, etc. | The **Payload Description** field shall contain user-provided information about the contents of the **Payload** field. This field can be left blank, but it can be useful when a user includes Generic Metadata payloads and wishes to help classify and identify such packages to systems into which AXF Objects are recovered. The examples given for this field apply to such Generic Metadata applications.<br><br>The encoding of the scalar values representing characters and ideographs shall be as specified in the **Payload Description Encoding Form** field. The correspondence between scalar values and the characters and ideographs that they represent shall be as defined in ISO/IEC 10646. |
| Payload Format Length | 110 + Payload Description Length | 2 | Unsigned 16-bit integer | 128 | The **Payload Format Length** field shall indicate the length of the **Payload Format** field expressed in bytes.<br>When the **Structure Identifier** value is AXF_OBJECT_FILE_PAYLOAD_START or AXF_OBJECT_FILE_PAYLOAD_STOP, the **Payload Format Length** shall have a value of zero. |

| Field Name | Byte Offset | Length (Bytes) | Data Type | Example | Description and Requirements |
|---|---|---|---|---|---|
| Payload Format | 112 + Payload Description Length | Variable length, defined by the **Payload Format Length** field | UTF-8 | application/xml | The **Payload Format** field shall indicate the format of the payload contained within the **Payload** field. The purpose of the **Payload Format** value is to assist in processing of the Binary Structure Container's payload.<br><br>The **Payload Format** field shall carry a comma (",")-separated list of one or more Media Types, as defined by RFC 6838, to specify a nested set of containers carrying the payload content. If only one Media Type is specified, it shall be that of the payload data. If more than one Media Type is specified, the Media Types shall specify the format of the payload containers, with the first Media Type specifying the outermost container and with each successive Media Type describing the format of that container's contents. The last Media Type in the list shall indicate the Media Type of the payload data.<br><br>The Media Type values used to define the **Payload Format** often include an implicit or explicit byte ordering specification, which, when present, shall specify the byte ordering of the payload contained within the **Payload** field. To the extent possible, Media Types that indicate byte ordering of the associated payloads should be used. If a Media Type value that indicates the payload format and the byte ordering of the payload is not available for a particular payload type or the byte ordering is unknown, then the Media Type value "application/octet stream," according to RFC 1494, should be used.<br><br>When the **Structure Identifier** value is AXF_OBJECT_FILE_PAYLOAD_START or AXF_OBJECT_FILE_PAYLOAD_STOP, the **Payload Format** field shall be omitted (i.e., have a length of zero). |

| Field Name | Byte Offset | Length (Bytes) | Data Type | Example | Description and Requirements |
|---|---|---|---|---|---|
| Payload Length | 112 + Payload Description Length + Payload Format Length | 8 | Unsigned 64-bit integer | 1a2b3c4d5e6f7a8b(h) | The **Payload Length** field shall indicate the size of the (possibly compressed) **Payload** expressed in bytes.<br><br>When the **Structure Identifier** value is AXF_OBJECT_FILE_PAYLOAD_START or AXF_OBJECT_PAYLOAD_STOP, the **Payload Length** field shall have a value of zero. |
| Payload | 120 + Payload Description Length + Payload Format Length | Variable Length Defined by the **Payload Length** field | As indicated by **Payload Format** | N/A | The **Payload** field shall contain the payload encapsulated by the Binary Structure Container.<br><br>When the **Structure Identifier** value is AXF_OBJECT_FILE_PAYLOAD_START or AXF_OBJECT_PAYLOAD_STOP, the **Payload** shall be empty and may be ignored by receiving applications. |
| Padding | 120 + Payload Description Length + Payload Format Length + Payload Length | Chunk Size – ((696 + Payload Description Length + Payload Format Length + Payload Length) Mod Chunk Size) | UTF-8 NUL characters | 0x00 | Padding added to align the entire data structure on Chunk-size boundaries defined for the particular AXF Object shall consist of 0x00 values.<br><br>Padding shall extend the Binary Structure Container by the minimum amount necessary to cause the last byte of the final field of the Binary Structure Container to appear in the byte position immediately preceding a Chunk boundary.<br><br>When the **Structure Identifier** value is AXF_OBJECT_FILE_PAYLOAD_START or AXF_OBJECT_FILE_PAYLOAD_STOP, the **Payload** will be empty, and Padding shall be applied to correctly position the last byte of the final field of the Binary Structure Container.<br><br>The byte length of the Padding is referred to as **Padding Length** and is used for Byte Offset Calculations of subsequent fields in the Binary Structure Container. |

| Field Name | Byte Offset | Length (Bytes) | Data Type | Example | Description and Requirements |
|---|---|---|---|---|---|
| Checksum Type | 120 +<br>Payload Description Length +<br>Payload Format Length +<br>Payload Length +<br>Padding Length | 16 | UTF-8 | SHA-1 | The **Checksum Type** field shall define the checksum algorithm used for structure verification. The **Checksum Type** value shall be drawn from the checksum algorithms specified in the list of examples in Section 10.21.1.3.<br><br>The following values shall be permitted:<br><br>CRC64<br>MD5<br>SHA-1<br>SHA-224<br>SHA-256<br>SHA-384<br>SHA-512 |
| Checksum | 136 +<br>Payload Description Length +<br>Payload Format Length +<br>Payload Length +<br>Padding Length | 512 | Unsigned 4096-bit integer | 12fe456d89c76562<br>5df9df9812fe456d<br>89c765625df9df98<br>3345678964ab500<br>000000 …. 0000(h) | The **Checksum** field shall carry the value of the checksum of the **Payload** field contained within the Binary Structure Container, determined using the algorithm specified in the **Checksum Type** field. It shall carry the value in the number of left-most bytes specified for the Checksum Type used and shall be padded in its right-most bytes with NUL values to a length of 512 bytes. |
| Structure Identifier 2 | 648 +<br>Payload Description Length +<br>Payload Format Length +<br>Payload Length +<br>Padding Length | 32 | UTF-8 | AXF_OBJECT_HEADER | The **Structure Identifier 2** field shall carry a duplicate copy of the value contained in the **Structure Identifier 1** field. |
| Chunk Size 2 | 680 +<br>Payload Description Length +<br>Payload Format Length +<br>Payload Length +<br>Padding Length | 8 | Unsigned 64-bit integer | 100000(h) | The **Chunk Size 2** shall indicate in bytes the Chunk size used for the current AXF Object. The same value is duplicated in the **Chunk Size 1** field. |

| Field Name | Byte Offset | Length (Bytes) | Data Type | Example | Description and Requirements |
|---|---|---|---|---|---|
| Structure Start Position | 688 +<br>Payload Description Length +<br>Payload Format Length +<br>Payload Length +<br>Padding Length | 8 | Signed 64-bit integer | 1a2b3c4d5e6f7a8b(h) | The **Structure Start Position** field shall specify the relative number of Chunks from the location of the **Structure Start Position** field to the first Chunk of the current Binary Structure Container. (Note: If the Binary Structure Container is contained in one Chunk, the value of **Structure Start Position** will be zero. Otherwise, the value of Structure Start Position always will be negative.)<br><br>Implementation note: SMPTE ST 2034-1:2014 specified the Data Type of **Structure Start Position** as an Unsigned 64-bit integer. AXF Objects created using that version will have positive values for **Structure Start Position**. On encountering positive values for **Structure Start Position**, AXF-aware applications compliant with this version should interpret them as negative values or zero to conform with the current definition of **Structure Start Position**. At the time that such AXF Objects are refreshed in storage, adjustment of the values to comply with the definition above is recommended. |

### 6.4.2 Media Data Structures

Each medium formatted according to this standard shall include specific data structures that define it as a valid AXF Medium. Depending upon the Storage Media Type with respect to Block-Based or File-System-Based media, certain structures are either required or optional (see Section 5.4 and Table 1).

One or more AXF Object may be stored on a single medium, and a single AXF Object may be stored on one or more media through use of the Spanning process (see Section 8). The layout of the various data structures on a medium is specific to the type of medium as described above. The general layout of AXF Objects on an AXF Medium is shown in Figure 3, which is representative of Block-Based Media – the most complex example.

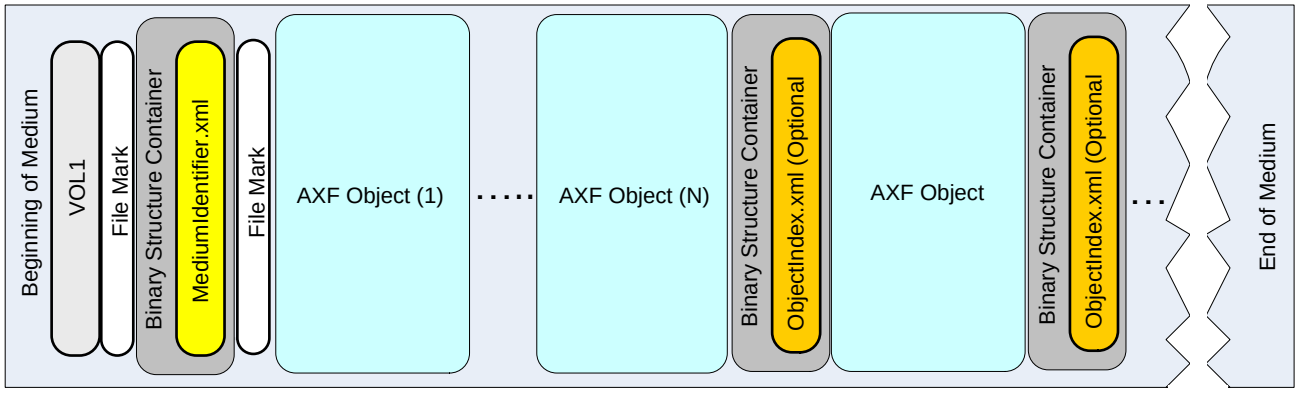



**Figure 3 – AXF Structure Layout on Block-Based Media**

The following sections detail AXF Media-specific structures. Section 5 defines which media data structures are applicable to each particular Storage Media Type.

#### 6.4.2.1 VOL1 Label

The VOL1 Label is an ISO/ANSI standard volume label (“VOL1”) structure as specified in ISO/IEC 1001:2012(E). This label is included to maintain compatibility with legacy applications for Storage Media Types that benefit from its use. The VOL1 Implementation Identifier indicates to applications with AXF compatibility that the medium has been properly prepared for AXF read/write activities. The VOL1 Label shall not be encapsulated inside a Binary Structure Container. The VOL1 Label shall be stored on a medium so that the first byte of the label is carried in the first byte of the first available block on the medium. Applications shall write the VOL1 Label structure when preparing a medium for AXF use and shall read the VOL1 Label**,** in combination with the AXF Medium Identifier (see Section 6.4.2.2), to detect whether the medium has been prepared for AXF Objects before any are stored.

To maintain compatibility with non-AXF applications, the VOL1 Label shall be followed by a device-specific File Mark on applicable media. The fields of the VOL1 Label are structured as defined in Table 3. See Table 1 for designations of the Storage Media Types on which the VOL1 Label shall be applied.

**Table 3 – Structural Elements of a VOL1 Label as Applied to AXF Media**

| Field Name | Byte Offset | Length (Bytes) | Data Type[1] | Example | Description |
|---|---|---|---|---|---|
| VOL1LabelIdentifier | 0 | 3 | ISO 646 | VOL | Identifies the label as a volume label. Per ISO/IEC 1001:2012(E), this field is set to "VOL" |
| VOL1LabelNumber | 3 | 1 | ISO 646 | 1 | Identifies the label as the first volume header label. Per ISO/IEC 1001:2012(E), this field is set to "1" |
| VOL1VolumeIdentifier | 4 | 6 | ISO 646 | ABC001 | Carries an identification of the volume |
| VOL1VolumeAccessibility | 10 | 1 | ISO 646 | Space | Indicates access restrictions. |
| (Reserved) | 11 | 13 | ISO 646 | Spaces | Per ISO/IEC 1001:2012(E), reserved for future standardization |
| VOL1ImplementationIdentifier | 24 | 13 | ISO 646 | AXF | Indicates the implementation that created the Volume Header Label. |
| VOL1OwnerIdentifier | 37- | 14 | ISO 646 | CreatorCompany | Indicates the owner of the volume. |
| (Reserved) | 51 | 28 | ISO 646 | Spaces | Per ISO/IEC 1001:2012(E), reserved for future standardization |
| VOL1LabelStandardVersion | 79 | 1 | ISO 646 | 4 | Indicates the version of the Label standard. Per ISO/IEC 1001:2012(E), this field is set to "4" for the 2012 version of the ISO/IEC 1001 standard. |

[1] A limited set of ISO 646 characters is permitted, described as "a-characters" in ISO/IEC 1001, Section 8.1.1 and Annex A.1.

The VOL1VolumeAccessibility field shall be set to a SPACE character to indicate unrestricted access, and the VOL1ImplementationIdentifier shall be set to "AXF".

#### 6.4.2.2 AXF Medium Identifier

An AXF Medium Identifier is an XML structure, carried as a Binary Structure Container Payload, that contains specific information relating to the AXF Medium itself. The AXF Medium Identifier shall be a Binary Structure Container Payload having a Structure Identifier value of AXF_MEDIUM_IDENTIFIER. The detailed XML structure of the AXF Medium Identifier can be found in Section 110.1.1 and in the XSD file associated with this document. The AXF Medium Identifier structure includes information key to the accurate recovery of AXF Objects contained on the AXF Medium, including a unique identifier, label, or human-readable identifier of the medium, as well as the block size used on the medium.

Refer to Section 5.4 and Table 1 for designation of the Storage Media Types on which the AXF Medium Identifier shall be applied.

On Block-Based media, the Medium Identifier shall be stored so that the first byte of the Binary Structure Container carrying the Medium Identifier is carried in the first byte of the first available block after the device-specific File Mark that follows the VOL1 label. The Medium Identifier shall itself be followed by a device-specific File Mark.

On File-System-Based media, virtualization of physical media is indicated by the presence of an AXF Medium Identifier within a File-System folder at a level lower than the root. The File-System folder containing the AXF Medium Identifier shall be the topmost folder of that virtual AXF Medium.

Note: This version of this standard does not specify the treatment of multiple partitions on a single physical Block-Based medium. Future revisions of this standard might do so.

#### 6.4.2.3 AXF Object Index

An AXF Object Index is an optional XML structure, carried as a Binary Structure Container Payload that contains a collection of the Object Footer structure data for AXF Objects contained on an AXF Medium. The information contained in an AXF Object Index is sufficient to recover and reconstruct the entire catalog of all AXF Objects listed within the AXF Object Index. In the case of a Block-Based Medium, an AXF Object Index contains a collection of Object Footer structures for all AXF Objects stored at locations prior to that at which the particular AXF Object Index is stored (i.e., at lower-ordered addresses on the medium). For a File-System-Based Medium, an AXF Object Index contains a collection of Object Footers for all AXF Objects stored on the medium and shall appear only once in the topmost folder of the AXF Medium. When an AXF Object Index is present on a File-System-Based medium, AXF-aware systems shall update and maintain it.

On Block-Based Media AXF Object Indices with higher-ordered addresses on the medium shall supplant those having lower-ordered addresses. AXF Object Index data only relates to AXF Objects preceding a particular AXF Object Index on a medium, and applications implementing AXF should ensure that AXF Objects stored on a medium following the last AXF Object Index on that medium are indexed along with those included in the last AXF Object Index. The **Structure Identifier** of a Binary Structure Container shall indicate carriage of an AXF Object Index in the Payload of the Binary Structure Container by having AXF_OBJECT_INDEX as its field value. The detailed XML structure of the AXF Object Index is given in Section 10.7 and in the XSD file associated with this document.

Although relevant and applicable in all scenarios and for all Storage Media Types, the use of AXF Object Index structures is most beneficial in cases of Block-Based Media. For File-System-Based Media, applications also can implement use of AXF Object Index structures, but, to reconstruct a medium's content index, it typically is just as fast for an application to scan the entire medium and then to process individually each of the Object Footer structures, except in special cases in which a File System is included on a Linear Medium. To facilitate reconstruction of the contents of any particular storage medium, it is not necessary for an application to include regular AXF Object Index structures, but doing so will assist in the speed of recovery operations, if they become necessary.

### 6.4.3 AXF Object Data Structures

Each AXF Object is a fully self-contained, encapsulated set of files, metadata, and any other ancillary information that adds context, relevance, or value to its contents. AXF is designed to handle encapsulation of a single file as easily as encapsulation of hundreds or millions of files. AXF Objects are structurally identical regardless of whether they are written to data tape, spinning disk, solid state media, or optical media – either Block-Based or File-System-Based.

#### 6.4.3.1 AXF Object Overview

Each AXF Object includes several data structures wrapped in Binary Structure Containers. These include an Object Header structure, any number of optional Generic Metadata structures, a File Payload Start structure, zero or more content files together with an associated AXF File Footer for each file, an optional File Payload Stop structure, and an Object Footer structure. Each AXF Object also can contain spanning information that relates the portion of the AXF Object stored on a particular medium with the remainder of that AXF Object stored on another medium or on other media and associative information that relates the AXF Object to other AXF Objects in a Collected Set. The overall structure of an AXF Object is diagrammed in Figure 4.

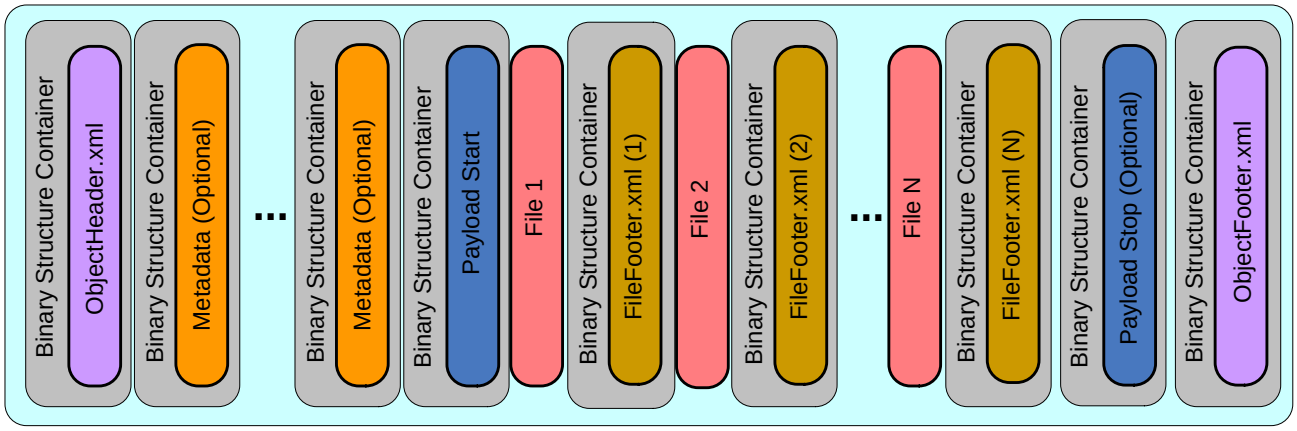



**Figure 4 – AXF Object Structure Overview**

#### 6.4.3.2 AXF Object Structural Components

Each AXF Object shall begin with an Object Header and end with an Object Footer. The Object Header and Footer structures contain descriptive XML metadata describing the contents of the AXF Object. Applications may use Object Headers and Object Footers as anchor points while parsing AXF Objects. Although following sections describe the general ordering of internal AXF structures, other structures can be expected to be embedded in these locations in future versions of this standard, and therefore applications should verify the Binary Structure Container type before processing each Binary Structure Container, ignoring unexpected or unknown Binary Structure Containers.

Following each Object Header may be zero or more Generic Metadata Containers. Each Generic Metadata Container shall comprise a Binary Structure Container, the payload of which is metadata associated with the AXF Object.

Following any Generic Metadata Containers shall be the File Payload portion of an AXF Object. The File Payload portion of an AXF Object shall begin with a File Payload Start structure. Immediately following the File Payload Start structure shall be any number of File Data + File Padding + File Footer triplets, the last of which may be followed by a File Payload Stop structure. File Padding shall be used to align the File Data on AXF Chunk boundaries to enable easy description of the locations and retrieval of the contents of the files contained within the AXF Object. Because the files in an AXF Object are not encapsulated in Binary Structure Containers, the File Payload Start and File Payload Stop structures enable locating the File Payload of an AXF Object while scanning at high speed.

The final Binary Structure Container of an AXF Object always shall be an Object Footer.

#### 6.4.3.3 File Tree

A File Tree structure shall be included in each Object Header and Object Footer, as they are more fully described below. Each File Tree structure shall include the names of the files and symbolic links contained in the related AXF Object File Payload; the paths, if any, through a folder structure to each of the contained files and symbolic links; and detailed information related to the folders, the files, the symbolic links, and their relationships to one another. The File Tree structure shall be expressed in XML, as described in detail in Section 10.14, and shall be composed of four data types: FileFolder, Folder, SymLink, and File, as described in Sections 10.10 through 10.13, respectively.

The File Tree within an AXF Object shall traverse from a root folder down a hierarchy within which all of the File Tree members shall be contained. The File Tree shall relate files, symbolic links, and the folders in which they are contained by establishing a path structure from the root folder to each contained folder, file, and symbolic link within the structure, as described in the XML listing within the File Tree of the associated AXF

Object. Folders never actually exist physically; they are represented only within File Tree listings and as items in path descriptions. Mechanisms are provided in the XML structure of the File Tree for indicating folders that are nested within other folders and folders that are peers with other folders and files and nested within a folder at a higher level in the folder hierarchy.

Index numbers shall be applied to each folder, symbolic link, and file in the File Tree of an AXF Object. The Index numbers within an AXF Object shall be scoped to the folders, symbolic links, and files contained within that AXF Object. Index numbering shall start with a value of '1' applied to the root folder of the File Tree. The Index number shall increment by one for each step in traversing the File Tree. The value of the final Index number in a File Tree shall equal the total number of files, symbolic links, and folders contained within the File Tree (including the root folder).

Along with each file listed in the File Tree structure, nested checksum type/value pairs, file identifiers of any type and number, and various file attributes also may be included. To promote compatibility across a broad spectrum of applications and technologies, multiple instances of checksums, identifiers, and attributes of different types may be associated with each file or symbolic link, both in the File Tree and elsewhere.

An example File Tree structure is shown in Figure 5. In that figure, on the left side, a hierarchical structure of folders, files, and symbolic links is shown in a file tree diagram. On the right side, the corresponding structure of files and symbolic links within an AXF Object is shown. On the left side, index numbers are given for folders, files, and symbolic links, as they would appear in FileTree structures, while on the right side the index numbers for only files and symbolic links are shown, as only the information related directly to files and symbolic links is included in File Footers and Symbolic Link Footers. The paths to all of the files and symbolic links are shown on both sides, as they would be expressed both in the FileTree structure and in the File Footer of each file or symbolic link. Paths are not explicitly expressed for folders in the File Tree structure; rather their presence is denoted by the XML structures naming them and assigning Index numbers to them. In the event of loss of all FileTree structures on a medium, the File Tree can be reconstructed by reading the paths in the File Footers and Symbolic Link Footers, in which case the existence of empty folders can be inferred from the Index numbering sequence, but the names of empty folders cannot be determined.

Note: The nature of the target of a symbolic link, i.e., a file or a folder, is not directly indicated by the data included in the Symbolic Link Footer. Determination of the nature of the target requires following the link and examining the data and/or structure found at the target of the link.

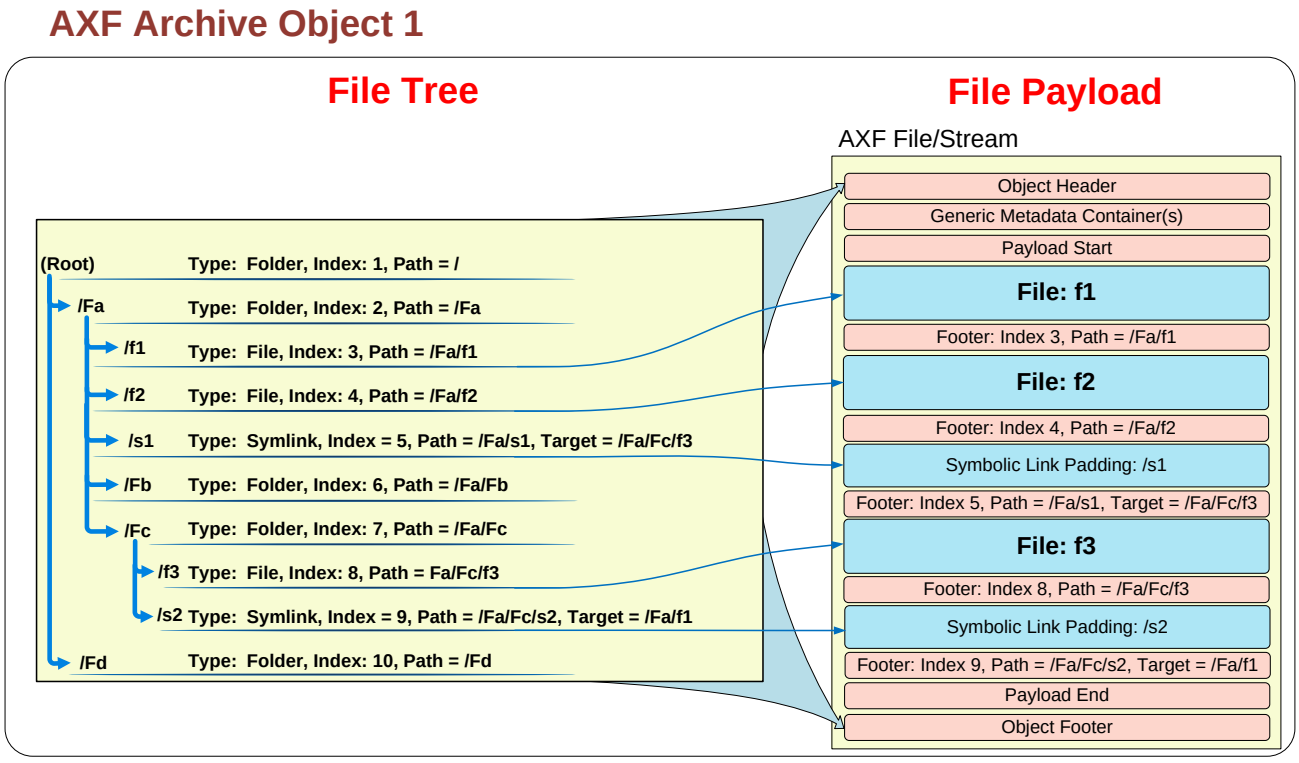



**Figure 5 – File Tree Structure Overview**

#### 6.4.3.4 Object Header Structure

An Object Header is an XML structure, carried as a Binary Structure Container Payload, that provides structural information about the AXF Object itself and information about the contents of the AXF Object, including such items as file names, file paths, checksum data when available, provenance information, and the like. Because an Object Header is written before much of the detailed information about the File Payload is known, much of the data that it contains is optional. Not optional are such details as the Chunk size of the AXF Object, and other data necessary to the accurate recovery of the AXF Object contents. To help enhance the robustness of the AXF Object, the Object Header also serves to provide redundancy of information carried in complete form in the corresponding Object Footer. In addition, data in Object Headers is used in the processing of linkages in Spanned Sets (see Section 8) and Collected Sets (see Section 9). The **Structure Identifier** of a Binary Structure Container shall indicate carriage of an Object Header in the Payload of the Binary Structure Container by having AXF_OBJECT_HEADER as its field value. The detailed XML structure of the Object Header shall be as given in Section 10.2 and in the XSD file associated with this document.

#### 6.4.3.5 Generic Metadata Container Structure

A Generic Metadata Container provides an optional, open, and extensible space for the direct association of free-form metadata with an AXF Object. It applies at the AXF Object level and can be used to carry metadata related to the AXF Object itself and/or to its contents. The metadata in such a container can be extracted from an AXF Object without having to restore any of the files contained in the associated File Payload.

Within a Binary Structure Container that denotes the carriage of generic metadata, the metadata may be in the form of an XML structure, plain text, binary data, or any other form of data expression. No specification is provided herein for the metadata carried within a Generic Metadata Container; any form of metadata described in other documents or privately defined may be carried in a Generic Metadata Container. Such information also may be stored in one or more files in the File Payload, but the generic metadata space is

provided for situations in which storage of metadata separate from the File Payload but carried with the associated AXF Object can be beneficial to particular applications. The **Structure Identifier** of a Binary Structure Container shall indicate carriage of Generic Metadata in the Payload of the Binary Structure Container by having AXF_OBJECT_METADATA as its field value.

In the case in which there is no generic metadata associated with an AXF Object, there shall be no Generic Metadata Container structures included within the AXF Object. Use of a Generic Metadata payload without additional definition of that payload, however, does not guarantee downstream usability. Different types of metadata associated with a single AXF Object should be carried in separate Generic Metadata Containers.

#### 6.4.3.6 File Payload Start Structure

The File Payload Start structure shall be an empty Binary Structure Container signifying that the Chunk immediately following the File Payload Start structure shall contain either the first portion (or all) of the first file of the AXF Object File Payload or an associated File Payload Stop structure (see Section 6.4.3.9). The File Payload Start structure may be used by applications that do not need to access other AXF structures but do need to perform any of a variety of operations on Payload Files. By locating the File Payload Start structure, an application can find the File Payload, which commences with the following Chunk. Using just the information stored in the associated File Footers, such an application can fully restore all Payload Files of an AXF Object, without the use of the Object Header or Object Footer. The **Structure Identifier** of a Binary Structure Container shall indicate the presence of a File Payload Start Binary Structure Container by having AXF_OBJECT_FILE_PAYLOAD_START as its field value.

#### 6.4.3.7 File and Symbolic Link Payloads

When present, each Content File shall be stored as a sequence of bytes, from beginning to end. Each Content File shall be stored with its first byte coinciding with the first byte of a Chunk. Each Content File shall occupy as many sequential Chunks as necessary to contain all of its data. A Content File shall not be wrapped in a Binary Structure Container. Each Content File shall be followed by sufficient File Padding to completely fill the last Chunk occupied by the Content File, thereby maintaining Chunk alignment of the File Payload structure. File Padding shall consist of bytes having a value of 0x00. Should a Content File fully fill an integer number of Chunks, then no File Padding shall be added. The final Chunk carrying a Content File (and any associated File Padding) shall be followed by a File Footer (see Section 6.4.3.8).

Byte ordering of File Payload files can be indicated in metadata related to the File Payload files, carried in multiple places within an AXF Object, in particular the related File Footer. See Section 6.2 for details.

The File Payload of an AXF Object may be spanned across multiple media, as described in Section 8. When spanning is applied, a Content File may be split into two or more fragments, the lower-ordered of which at each span point will be terminated with a Fragment Footer and the higher-ordered of which at each span point will be preceded by an Object Header followed by a Fragment Header. In such instances, the sequential storage of the Content File in Chunks shall bridge across as many media as necessary to fully contain the Content File.

When a Symbolic Link is included in a FileTree to reference a file or a folder using an alias, a single chunk of padding data (called a “Padding Chunk”) shall be inserted into the File Payload to enable use of a footer structure for the Symbolic Link equivalent to a File Footer. To enable operation of the FileTree path structure with respect to the Symbolic Link, the Padding Chunk shall be named in the same way that files are named. All bytes in the Padding Chunk shall have a value of 0x00. The Padding Chunk for a particular Symbolic Link shall be followed by a related Symbolic Link Footer structure, as described in Section 6.4.3.8.

#### 6.4.3.8 File and Symbolic Link Footer Structures

A File Footer shall be an XML structure, carried as a Binary Structure Container Payload, written following each Content File in a File Payload. A File Footer structure contains detailed information regarding the associated Content File, including the size of the associated Content File and an optional file-level checksum,

designed to be processed by an application on the fly during restore operations to ensure bit fixity. The **Structure Identifier** of a Binary Structure Container shall indicate carriage of a File Footer in the Payload of the Binary Structure Container by having AXF_FILE_FOOTER as its field value. The detailed XML structure of a File Footer shall be as given in Section 10.4 and in the XSD file associated with this document; in turn, it depends on the **FileFolder** XML structure described in Section 10.10 and also in the XSD file.

A Symbolic Link Footer shall be an XML structure, carried as a Binary Structure Container Payload, written following each Symbolic Link Padding Chunk in a File Payload. A Symbolic Link Footer structure contains detailed information regarding the Target of the Symbolic Link. The **Structure Identifier** of a Binary Structure Container shall indicate carriage of a Symbolic Link Footer in the Payload of the Binary Structure Container by having AXF_FILE_FOOTER as its field value. The detailed XML structure of a Symbolic Link Footer shall be as given in Section 10.4 and in the XSD file associated with this document; in turn, it depends on the **Symlink** XML structure described in Section 10.13 and also in the XSD file.

#### 6.4.3.9 File Payload Stop Structure

The File Payload Stop structure shall be an empty Binary Structure Container signifying that the previous Chunk contains the File Footer of the last file in the AXF Object File Payload. The File Payload Stop structure can be used by applications that do not need to access other AXF structures but do need to perform a variety of operations on Payload Files. By locating the File Payload Stop structure, an application readily can find the end of the File Payload. The File Payload Stop Structure provides symmetry with the File Payload Start Structure, thereby enabling an application to locate the bounds of the AXF Object File Payload. Using just the information in the associated File Footers, such an application can fully restore all Content Files contained in the File Payload of an AXF Object, without relying on the use of the Object Header or Object Footer. The **Structure Identifier** of a Binary Structure Container shall indicate the presence of a File Payload Stop Binary Structure Container by having AXF_OBJECT_FILE_PAYLOAD_STOP as its field value.

#### 6.4.3.10 Object Footer Structure

An Object Footer may be used in similar fashion to an Object Header, and can be used by applications to locate and process the files and other information contained within the AXF Object. Each Object Footer shall contain a collection of metadata that describes the contents of the AXF Object. The Object Header and Object Footer data structures are fundamentally identical; however, some fields that are optional in the Header are mandatory in the Footer. The **Structure Identifier** of a Binary Structure Container shall indicate carriage of an Object Footer in the Payload of the Binary Structure Container by having AXF_OBJECT_FOOTER as its field value. The detailed XML structure of the Object Footer shall be as given in Section 10.6 and in the XSD file associated with this document.

Object Footers serve two purposes: They provide updated information about the contents of an AXF Object that was not available when its Object Header was created, and they provide redundancy by duplicating specific information from the corresponding Object Header and from the File Footers of all of the files contained within the AXF Object. In addition, data in Object Footers may be used in the processing of linkages in Collected Sets, as explained later in Section 9.

#### 6.4.3.11 Fragment Footer Structure

AXF Object Fragment links consist of Fragment Footer and Fragment Header pairs that carry pair-bond identifiers when an AXF Object is spanned across multiple storage media. A Fragment Footer is an XML structure, carried as a Binary Structure Container Payload that provides linkage information between two fragments of an AXF Object. A Fragment Footer shall be inserted at the end of a section of File Payload that is being spanned to a following medium in a Spanned Set. The Fragment Footer shall occupy the last active Chunk on a medium; there shall be no Object Footer included at the end of a section of File Payload that does not complete the File Payload of the Spanned Set. The Fragment Footer shall include a Fragment Pair UUID value matching the Fragment Pair UUID value carried in the corresponding Fragment Header that begins the File Payload section carried on the immediately following, higher-ordered medium in the Spanned Set. The **Structure Identifier** of a Binary Structure Container shall indicate carriage of a Fragment Footer in

the Payload of the Binary Structure Container by having AXF_OBJECT_FRAGMENT_FOOTER as its field value.The detailed XML structure of the Fragment Footer shall be as given in Section 10.5 and in the XSD file associated with this document.

#### 6.4.3.12 Fragment Header Structure

AXF Object Fragment links consist of Fragment Footer and Fragment Header pairs that carry pair-bond identifiers. A Fragment Header is an XML structure, carried in a Binary Structure Container Payload, that provides linkage information between two fragments of an AXF Object. A Fragment Header shall be inserted at the beginning of a section of File Payload that has been spanned from a previous medium in a Spanned Set. The Fragment Header shall follow an Object Header that duplicates the Object Header(s) carried in the lower-ordered fragment(s) in the Spanned Set. The Object Fragment Header shall include a Fragment Pair UUID value matching the Fragment Pair UUID value carried in the corresponding Fragment Footer that ended the File Payload section carried on the immediately preceding, lower-ordered medium in the Spanned Set. The **Structure Identifier** of a Binary Structure Container shall indicate carriage of a Fragment Header in the Payload of the Binary Structure Container by having AXF_OBJECT FRAGMENT_HEADER as its field value. The detailed XML structure of the Fragment Header shall be as given in Section 10.3 and in the XSD file associated with this document.

# 7 General Usage Considerations

General usage considerations include methods such as file naming conventions necessary to the interchange of stored content and practices intended to assure fast and reliable retrieval of such content. They apply when creating, reading, writing, parsing, recovering, or indexing AXF Objects and/or AXF Media.

## 7.1 File Naming

When stored on File-System-Based Media, files containing AXF Objects and files associated with the AXF protocol shall be identified with specific file name extensions. The file name extensions shall be applied as specified in Table 4.

**Table 4 – AXF Protocol Filename Extensions**

| AXF Protocol Component | Filename Extension | Example Filenames |
|---|---|---|
| AXF Object | .axf | *Object UUID*.axf |
| AXF Object Index | .axfi | *Medium UUID*.axfi |
| AXF Medium Identifier | .axfm | *Medium UUID*.axfm |

The Base Names of files containing AXF Objects, AXF Object Indices, and AXF Medium Identifiers are not constrained by this standard. Nevertheless, the Base Names of the files containing the associated structures should be the UUIDs that are carried in those structures to uniquely identify them. For those character sets where applicable, both Base Names and extensions shall be treated in a case–insensitive manner by readers of the structures; i.e., fully upper case, fully lower case, and mixed case names having the same alphabetic characters shall be interpreted as the same names.

## 7.2 Media Preparation

Media Preparation shall include the addition of an AXF Medium Identifier structure to the medium and also shall include the addition of a VOL1 label (see Section 6.4.2.1) prior to the AXF Medium Identifier as indicated in Table 1 — Relationships between AXF Structures and Storage Media Types.

Note: The absence of their formatting as AXF Media does not preclude the storage of AXF Objects on certain Storage Media Types.

Despite the requirement for media preparation to enable the medium identification functionality of the AXF protocol, AXF-aware systems retrieving AXF Objects from media are expected to recover AXF Objects despite the absence of, or corruption of, an AXF Medium Identifier; e.g., when that portion of a medium has become damaged.

The AXF Medium Identifier shall be written by an application when preparing a medium for first-time use or subsequent reuse, followed by a device specific File Mark, as applicable (see Table 1). It generally is not expected that the AXF Medium Identifier will be updated by an application following the preparation of a medium for AXF utilization, except in the case in which the medium has been erased and is being prepared again for use as an AXF Medium. Although some storage media allow easy random access to the AXF Medium Identifier structure, it should not be updated once created.

Physical media can be subdivided into multiple virtual media, when supported by the particular storage technology used (e.g., partitions or sub-directories). In such cases, multiple, virtual AXF Media can appear on individual physical media. A virtual AXF Medium shall be created by placing an AXF Medium Identifier at the beginning of a partition or in a sub-directory. In an instance of a virtual AXF Medium, the structure that contains the AXF Medium Identifier shall serve as the top level of the virtual AXF Medium.

### 7.2.1 Block-Based Media

Block-based media shall be prepared for use as AXF Media prior to first utilization. Such preparation shall consist of placing VOL1 labels and AXF Medium Identifiers on the media, as specified in Sections 6.4.2.1 and 6.4.2.2. Once AXF Objects are stored on media, it can become quite difficult to insert VOL1 labels and Medium Identifiers after the fact, leading to the requirement for initial preparation of block-based media.

### 7.2.2 File-System-Based Media

On File-System-Based Media, medium preparation operations shall include creation of AXF Medium Identifiers and their placement at the root locations of the storage media being prepared. It is possible for different AXF-aware applications to have different root entry points into the file structure of a storage medium, resulting in multiple Medium Identifiers being placed on a medium. Consequently, during the media preparation process on file-system-based media, AXF-aware systems shall not remove or modify any AXF Medium Identifiers found elsewhere on storage media unless done intentionally as part of the media preparation process.

## 7.3 AXF Object Index Structures

AXF Object Indices are optional structures that can be used to assist in the rapid recovery and indexing of AXF Media and the AXF Objects they contain. Such structures shall be supported on both Block-Based and File-System-Based Media.

AXF Object Index structures are defined in Section 6.4.2.3 and include collections of Object Footer data for all AXF Objects contained on Media prior to the locations at which the AXF Object Indices are stored in the case of Block-Based Media and for all AXF Objects contained on the Media for File-System-Based Media. AXF-aware applications can utilize AXF Object Index structures to speed the indexing of Media when those Media are accessed without prior knowledge of their contents by the applications.

On File-System-Based Media, AXF Object Indices (if used) shall be updated following each successful write or delete operation performed, allowing for immediate rebuilding by an application of its internal index simply by reading an AXF Object Index structure. Since such a Medium might have been written by non-AXF-aware applications, either adding non-AXF files or deleting AXF Objects, an application should rely upon AXF Object Indices only to provide a quick view of all AXF Objects and their contents contained on a given Medium. For certainty about the AXF Objects contained on the Medium, the application should scan all files contained on the Medium, determine whether each represents a valid AXF Object, and reconcile Object Footer structures to provide a validated view of the AXF Objects and the AXF Object contents contained on the Medium.

In the case of a Block-Based Medium, to ensure that other AXF-aware applications can quickly and easily index it and to provide maximum redundancy of critical data structures, an AXF-aware application should write a current AXF Object Index structure on the medium periodically during the addition of AXF Objects to it and just prior to transporting it. Although not necessary to enable indexing of and access to stored AXF Objects and their content by other applications, inclusion of current AXF Object Indices on AXF Media helps to assure quick and reliable retrieval of AXF Objects and the files that they contain.

## 7.4 Creating, Reading, Writing, Copying, and Transferring AXF Objects

AXF features only can be fully utilized in cases in which applications performing operations have implemented the AXF protocol. It is possible, however, for some applications to complete certain operations on AXF Objects and AXF Media with no awareness of AXF. Many of the benefits of this standard can be lost in such cases, but they do allow for wider application of AXF in situations in which the AXF protocol is not fully implemented.

### 7.4.1 AXF-Aware Applications

During copy and move operations, AXF-aware applications shall unpack and repack AXF Objects. These processes allow the updating of important provenance metadata, realignment with destination Media Block boundaries, updating of Chunk and Block relative and absolute position information, and validation of all File Payload data and AXF Object data structures to ensure that bits remain unaltered while streaming. During these operations, AXF-aware applications shall ensure the updating of Object Header metadata to align with that contained in corresponding Object Footers, assuring agreement within the structure pairs. An AXF-aware application shall determine whether a destination Medium has been prepared as an AXF Medium and whether AXF Object Index structures have been enabled and maintained on it. If they have, the AXF Object Index on the target Medium shall be updated to indicate the existence of all newly added AXF Objects following a successful copy or move operation.

AXF-aware applications shall ensure that proper reference and pointer information is maintained for each AXF Object and AXF Media data structure during all creation, copy and move operations. AXF-aware applications shall validate this information during read operations to ensure its integrity and shall attempt to recover inconsistent information automatically or shall provide notification in cases where unrecoverable inconsistencies or other issues are detected.

### 7.4.2 Non-AXF-Aware Applications

By design, applications that have not implemented the AXF protocol are permitted to copy and move AXF Objects when File-System-Based Media are involved. This is possible because each AXF Object is fully self-contained, is deterministically constructed, and does not mandate Chunk alignment on File-System-Based Media to permit subsequent access, recovery, copy, or move operations by both AXF-aware and -unaware applications.

Although not absolutely necessary, by intentional design to facilitate wider applicability of the AXF protocol, applications should ensure that provenance metadata and some of the resiliency characteristics (such as AXF Object Index structures) are employed and maintained.

Applications that have not implemented the AXF protocol can read AXF Objects contained on Block-Based Media, but Block and Chunk data need to be known *a priori* as these cannot be parsed from the AXF Media or AXF Object data structures without awareness of the AXF protocol. It is not possible for non-AXF-aware applications to write compliant AXF Objects to Block-Based Media, as key position data internal to the AXF Objects have to be updated to allow subsequent indexing and recovery of File Payload data by other systems.

Any application can delete AXF Objects from File-System-Based Media. For Block-Based Media, deletion typically involves removal of database records or pointers to data on the Media that contain the AXF Objects

but not deletion of actual AXF Objects data. Thus, for Block-Based Media, short of fully erasing the Media, it only is possible for any application to remove references to AXF Objects, effectively deleting them, but those AXF Objects will reappear on the Media should it be reindexed without knowledge of prior deletions.

### 7.5 Nesting AXF Objects

The AXF protocol fully supports nested AXF Objects, which are AXF Objects contained within other AXF Objects. Such inclusion is possible because each constituent AXF Object data structure contains the UUID assigned to the AXF Object of which it is a component. This UUID inclusion allows an AXF-aware application to identify specifically which structures are components of the AXF Object currently being processed and to ignore other structures that might be part of Generic Metadata Container or File Payload data of another AXF Object either that it contains or that contains it.

## 8 Spanning

Spanning is a segmentation and reassembly process for storing AXF Objects across multiple media, providing the necessary linkages to reconnect with one another the fragments of a spanned AXF Object (stored on a set of Media known as a “Spanned Set”).

There is no limit to the number of Media and there is no limit to the types or generations of Media that can be employed to store a spanned AXF Object.

The AXF spanning structure establishes linkages between the segments of a spanned AXF Object and between the storage media containing the spanned AXF Object, as storage of the spanned AXF Object moves from one Medium to the next. An AXF-aware application shall use the linkage data stored within the AXF spanning structures to reassemble the AXF Object from its spanned segments.

Segments of AXF Objects are identified as “Fragments” in the definition of Spanning. Fragments have headers and footers, as described below. Fragment footers, in particular, have been designed to take up no more than a single Block on any Medium (or on Media having moderate- or larger-sized Chunks, no more than a single Chunk) so that they can be placed in the remaining space at the end of a Medium when it becomes apparent to the system writing to the Medium that the Medium capacity is about to run out.

### 8.1 Spanning Linkages

To connect the fragments in a Spanned Set, a number of linkages are provided at each span point in the Spanned Set. The linkages consist of shared **FragmentPairUUID** values that connect successive fragments in the Spanned Set.

To connect Media, the Medium preceding a Span Point may include, in the Fragment Footer at the end of the Medium, the UUID that identifies the next Medium in the Spanned Set. Similarly, the Medium following a Span Point may include, in the Fragment Header at the start of the Medium, the UUID that identifies the preceding Medium in the Spanned Set.

The first Fragment in a Spanned Set shall not contain a Fragment Header — only a Fragment Footer. Correspondingly, the last fragment in a Spanned Set shall contain a Fragment Header but not a Fragment Footer. Each fragment in a Spanned Set shall begin with an Object Header, which shall precede the Fragment Header when it is present.

To connect the Fragments of a spanned AXF Object, each Fragment shall contain, in the Object Header that begins it, a single **AXF Object UUID** that identifies all of the Fragments of the AXF Object. Additionally, the Fragment Footer at the end of a Fragment preceding a Span Point and the Fragment Header at the start of the fragment following the Span Point each shall contain a shared UUID (the Fragment Pair UUID) that identifies the pair of Fragments that abut the Span Point. Each Fragment also shall be given a sequential number that identifies its position in the Spanned Set (the Fragment Number). These relationships and their connections are shown in Figure 6.

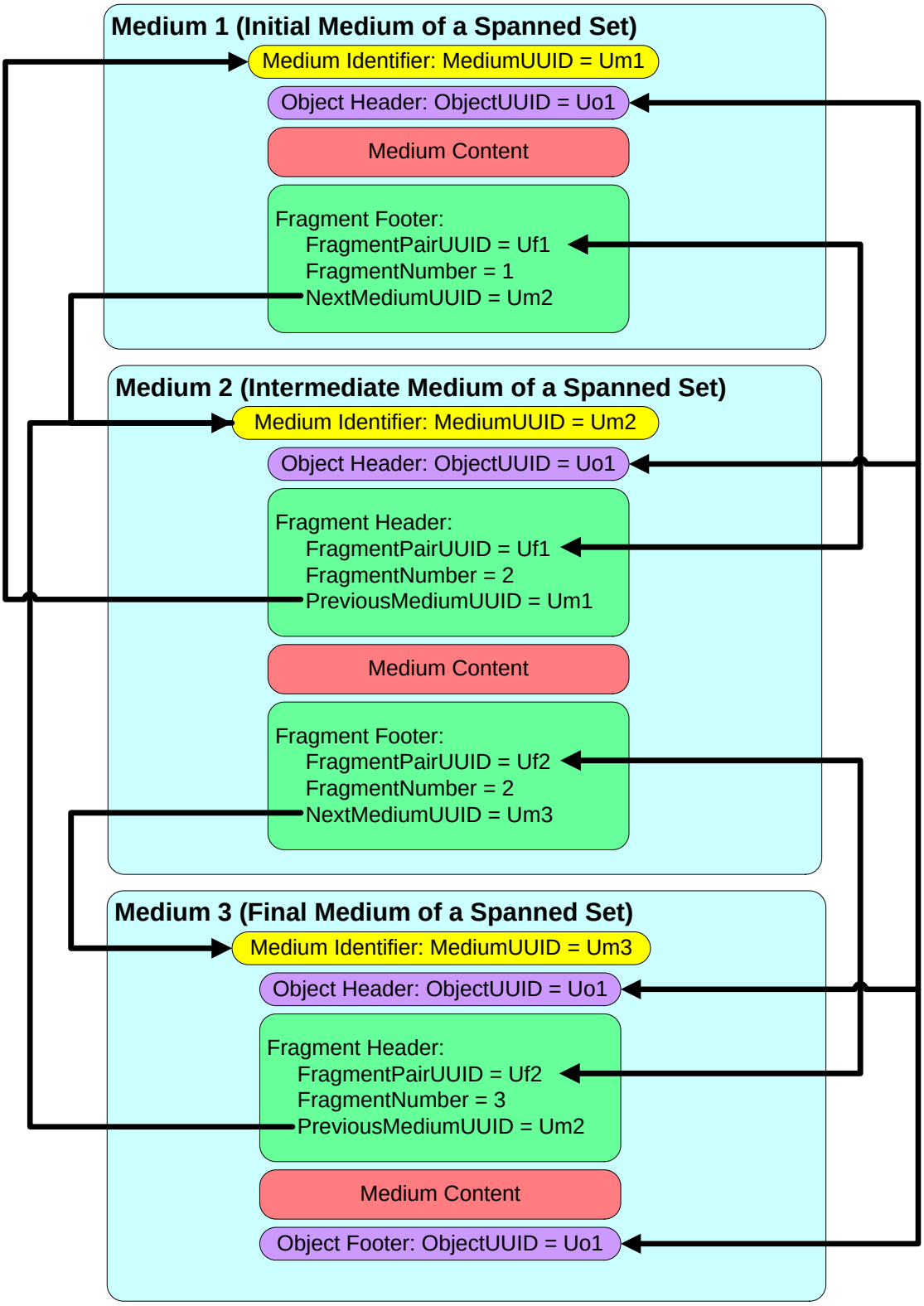



**Figure 6 – Linkages between Fragments of a Spanned Set**

A more complete depiction of the Media, AXF Objects, files, and structures involved in a Spanned Set is provided in Figure 7. It shows three Media and represents the structure when three or more media are included in a Spanned Set. In Figure 7, files #2 and #4 reached the ends of their Media. Both files were spanned to other Media to complete their storage.

When only two Media are included in a Spanned Set, the structures used are those only of Medium 1 and Medium 3, as enumerated in both Figure 6 and Figure 7.

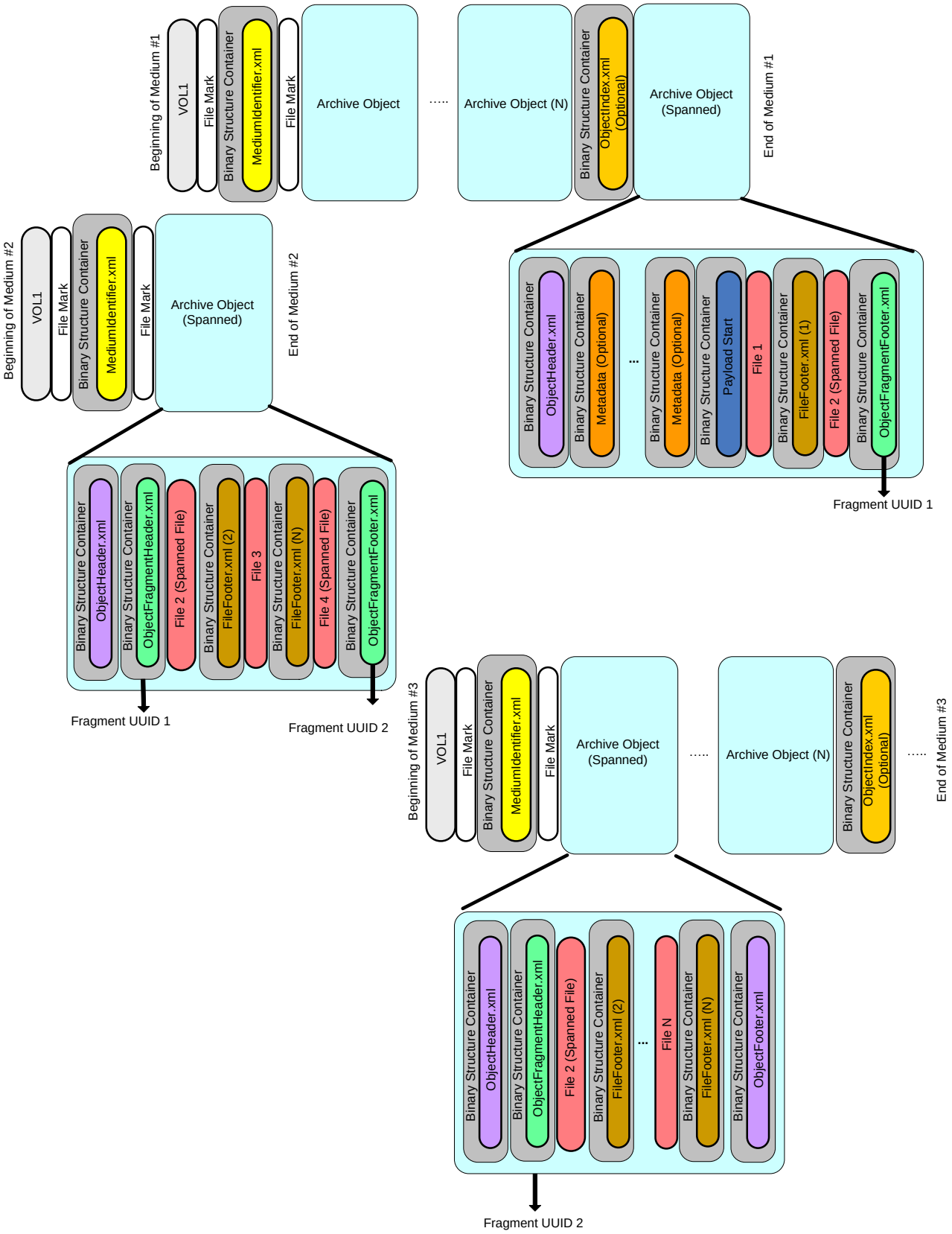



**Figure 7 – Multiple Medium Spanning**

## 8.2 Encountering a Spanning Situation

An application creating an AXF Object has the option either to keep track itself of the writing location on the Medium as the AXF Object is being written to the Medium or to allow the Medium to notify the application when the Medium is at full capacity. If the application relies on the Medium to indicate full capacity, then the application has to instruct the Media drive to back up two blocks from the end of the Medium to position the writing of the fragment footer at the end of the Medium.

When an application determines that a Media span is necessary, linkage is necessary between the current and next Media. The first decision for the application is to choose the next Medium type. Once that is determined, a Fragment Header and Fragment Footer pair are built to provide the linkage. The connections between the header/footer pair were shown in Figure 6 above for a 3-or-more-medium Spanned Set. When only two media are involved in a Spanned Set, the formats of Medium 1 and Medium 3 are used together, with the format of Medium 2 omitted from the Spanned Set.

## 8.3 Recovery of Spanned AXF Objects

When an application recovering an AXF Object encounters a Fragment Footer, the presence of the Fragment Footer indicates that the current AXF Object is spanned to another Medium. The application shall use the information provided in the Fragment Footer to determine the identity of the succeeding Medium as well as several linkages to the next Fragment.

Note: The next Fragment might have been stored on a Storage Media Type different from that carrying the current fragment. It is the responsibility of the system managing recovery of the AXF Object or of the end user to enable the application to access succeeding Fragments on different Storage Media Types.

For systems having Media already registered in their databases, it is the responsibility of those systems to control drive loading with succeeding Media in Spanned Sets and to manage reassembly of the Fragments of the spanned AXF Objects into their original forms.

For systems importing AXF Objects into their databases, it is the responsibility of those systems to identify succeeding Media containing subsequent Fragments in Spanned Sets and to provide needed indications to operators or to automated loaders so that the media can be made available when they are needed. This process can be aided by inclusion of a mechanism for pre-reading Media for identification and database registration purposes.

## 8.4 Spanning Rules

The first structure of an AXF Object shall be an Object Header. This requirement also shall apply to the Fragments of Spanned Sets.

When spanning is necessary, the last structure in a Fragment at the end of a medium shall be a Fragment Footer, and the next Fragment in the Spanned Set shall begin with an Object Header followed by a Fragment Header.

Fragments shall not be created without inclusion of Fragment Footers and linked Fragment Headers at the ends and beginnings, respectively, of successive fragments.

Note: This rule explicitly prohibits creation of Fragments during copying without following the Spanning procedures defined herein. Splitting a Fragment into smaller pieces during a copying process, without application of the necessary formatting elements, will result in some or all of the data following and possibly preceding the new split point becoming irretrievable.

File Footers shall be contained on the same media as the ends of their associated files; i.e., files never shall be separated from their associated File Footers by a span, with the file on one medium and its File Footer on the next.

If a File and its File Footer will not fit on a single Medium, the file itself shall be spanned across two Media, so that the File Footer always occupies the block immediately following the last byte of the file.

Any number of Fragments shall be permitted in a Spanned Set.

Fragments may start at locations on a Medium other than the beginning. Fragments may end at locations on a medium other than the end.

Multiple Fragments from a single Spanned Set may be stored on the same Medium. An example of this situation can occur when Fragments originally stored on separate Media are copied to a single Medium without a defragmentation process.

Note: Although permitted, this practice is not recommended.

# 9 Collected Sets

A Collected Set is a structure that enables adding files to, replacing files in, and deleting files from an AXF Object created at an earlier time. A Collected Set shall comprise an Anchor Object, having a **CollectedSetSequence** value of 1, and one or more subsequent AXF Objects having higher **CollectedSetSequence** values. A Product Object shall comprise the compilation of the Anchor Object of the Collected Set and subsequent AXF Objects, applied sequentially according to their **CollectedSetSequence** values. AXF Objects added to a Collected Set subsequent to the Anchor Object shall modify the complement of files included in the Product Object that results from combining the AXF Objects of the Collected Set having lower **CollectedSetSequence** values than the AXF Object being added. The Collected Set structure provides the necessary linkages to connect its AXF Objects with one another to produce a different complement of files within the Product Object than initially existed in the Anchor Object. The AXF Objects in a Collected Set shall have a specific order, with AXF Objects in the Collected Set having higher **CollectedSetSequence** values being read after AXF Objects having lower **CollectedSetSequence** values. As a result of the accumulation of contents and processing instructions of the sequence of subsequent AXF Objects, the contents of the Product Object produced by the Collected Set are altered with the addition of each successive AXF Object to the sequence in the Collected Set.

There is no limit to the number of AXF Objects that can be included in a Collected Set.

Linkages between the AXF Objects in a Collected Set are established by Object Header and Object Footer structures. During compilation of the Product Object of a Collected Set, the application reading the AXF Objects in the Collected Set, using the linkage data, mounts media in the order necessary to assemble the Collected Set and applies the processes indicated for its constituent files. This assembly procedure includes making changes to the **FileTree** structure, adding, replacing, and deleting files, as necessary.

## 9.1 Collected Set Linkages

To support assembly of the files in a Collected Set into a final Product Object, a number of linkages are provided in each Object Header and Object Footer in the Collected Set. The linkages shall consist of a shared UUID value that connects AXF Objects in the Collected Set plus **CollectedSetSequence** values to place successive AXF Objects in sequential order. The shared UUID value shall be the **ObjectUUID** of the first AXF Object in a Collected Set, which UUID value shall be duplicated in the **CollectedSetUUID**s of all AXF Objects that are members of the same Collected Set.

## 9.2 Collected Set Structure

To enable modification of AXF Objects after their creation in an Anchor Object, subsequent AXF Objects in a Collected Set shall contain instructions as to what is to be done with respect to individual files contained in AXF Objects in the Collected Set having lower **CollectedSetSequence** values, and those subsequent AXF Objects also may contain files to be added to the Product Object of the Collected Set or to be substituted for files previously included in the Product Object.

Figure 8 shows the relationships between AXF Objects in a Collected Set. As seen in the Object Header and Object Footer structures, each AXF Object has its own AXF **ObjectUUID** for identification. The **CollectedSetUUID** in the Anchor Object shall equal the AXF **ObjectUUID** of that AXF Object, and the **CollectedSetSequence** value of the Anchor Object shall be set to 1. Subsequent Objects in the Collected Set (those having higher **CollectedSetSequence** values than 1) shall have **CollectedSetUUID** values equal to the **CollectedSetUUID** (and AXF **ObjectUUID**) value of the Anchor Object, as indicated by the arrow in Figure 8, and those Subsequent Objects shall have **CollectedSetSequence** values starting at 2 and incrementing by 1 from one to the next in sequence in the order of their creation.

## 9.3 Add/Replace/Delete Processes

Three processes are defined for modifications of Product Objects that can be made by Subsequent Objects in a Collected Set. They are ADD, REPLACE, and DELETE. ADD means that the file to which it applies shall be added to the Product Object of the Collected Set, without deletion of a corresponding file. Such added files shall be uniquely identified within the scopes of the Collected Sets within which they reside. REPLACE means that the file to which it applies shall be added to the Product Object of the Collected Set, while a corresponding file having identical identification parameters shall be removed from the preceding Product Object when creating the next Product Object. DELETE means that the file to which it applies shall be removed from the preceding Product Object when creating the next Product Object. The preceding Product Object is the one that results from the compilation of all members of the Collected Set having CollectedSetSequence values lower than the CollectedSetSequence value of the AXF Object in which the process instructions are found. The next Product Object is the one that results from the compilation of all members of the Collected Set having CollectedSetSequence values lower than the CollectedSetSequence value of the AXF Object in which the process instructions are found with inclusion in the compilation of the AXF Object in which the process instructions are found.

Figure 8 portrays an example of making modifications to the contents of an AXF Object. When first created, the upper AXF Object in the figure is a standalone AXF Object, but nevertheless its **CollectedSetUUID** is matched to its AXF **ObjectUUID**. When it becomes desirable to modify the upper AXF Object, another AXF Object (the lower one in Figure 8) is created and given the same **CollectedSetUUID** value as the upper AXF Object. At the time of creation of the second AXF Object, the first AXF Object becomes the Anchor Object for what is now a Collected Set, and the **CollectedSetSequence** values of the second and all following AXF Objects are numbered sequentially.

The Anchor Object in Figure 8 contains one Video file, Video1, and two Audio files, Audio1 and Audio2. With the addition of the Subsequent Object to the Collected Set, the new audio file Audio1 contained in the second AXF Object is marked as a replacement for the Audio1 file that is contained in the Anchor Object, audio file Audio2 is marked as deleted from the Product Object per the instructions in the second AXF Object, and closed caption file Closed Caption1, also contained in the second AXF Object, is added to the Product Object of the Collected Set.

Note: When a file is marked as deleted in a Subsequent Object, the file is not physically removed from the original AXF Object in which it was contained; rather it is removed from inclusion in the Product Object when it is compiled, as described in Section 9.

## 9.4 Tracking Versions

When files are removed from the Product Object of a Collected Set, either through replacement or deletion, they are not physically removed from the AXF Objects in which they are contained. Instead, in response to instructions included in one or more Subsequent Objects in the Collected Set, they are deleted from the compilation process that assembles the Product Object. As a consequence of this method, it is possible to compile earlier Product Object versions from a Collected Set by applying only the Anchor Object and the sequence of Subsequent Objects through and including the Subsequent Object that completed the version of interest. In such cases, the **CollectedSetSequence** value serves as a version number for the Collected Set.

When multiple members of a Collected Set are created over time, they can be stored on a multiplicity of Media. Later, it can be desirable to combine the members of the Collected Set onto a single Medium while retaining the Collected Set members to permit recompilation of any of its versions. Retention of multiple versions of Collected Sets can be facilitated in such instances through use of nesting of AXF files, which are the containers of AXF Objects, as described in Section 7.5 Nesting AXF Objects .

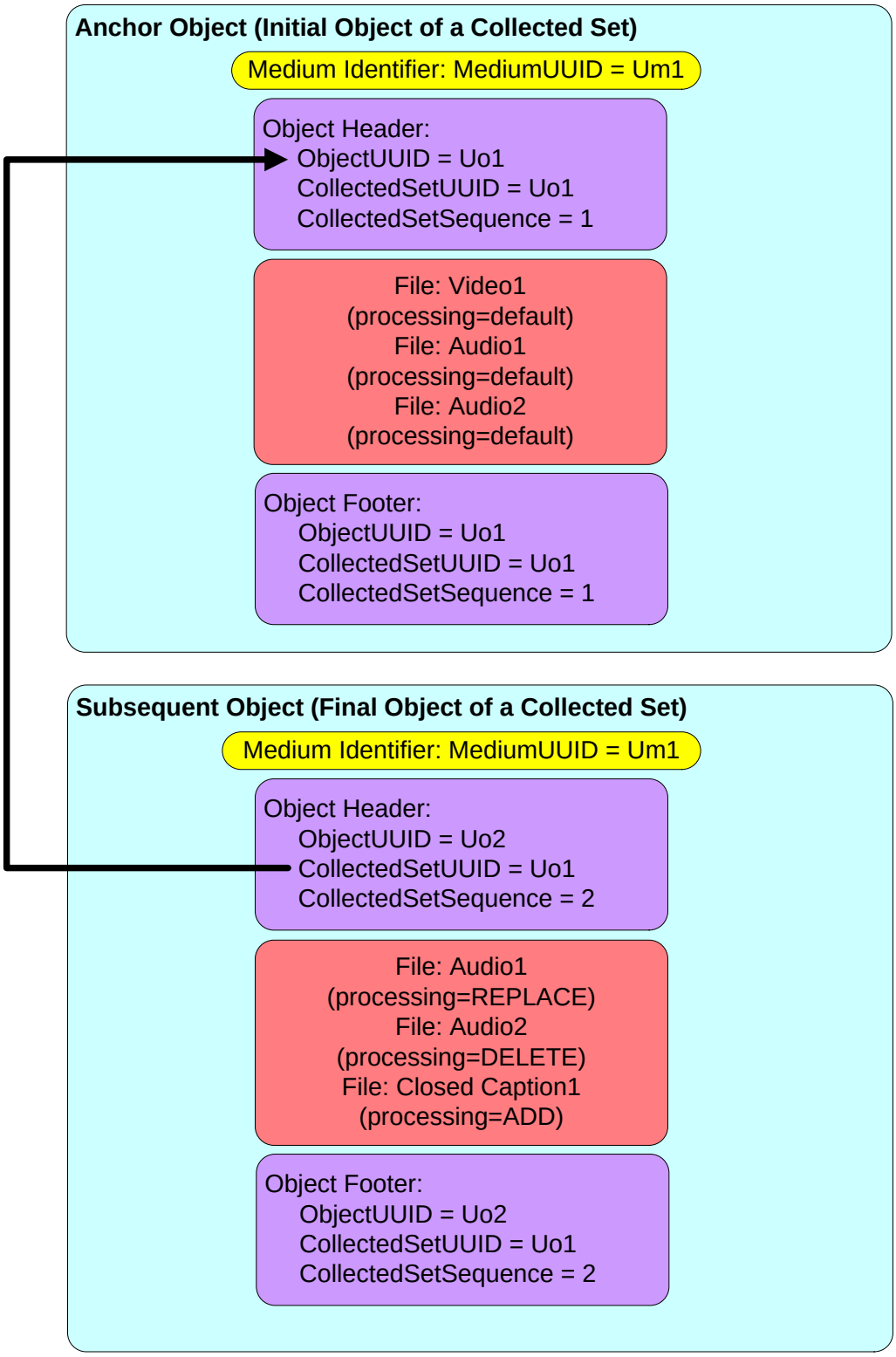



**Figure 8 – Linkages and Processing Instructions in Collected Sets**

# 10 AXF Data Model

The following subsections delineate the XML constructs that comprise the various structures of AXF-formatted media and AXF Objects. Each major subsection describes one structural element of the AXF system and includes a UML diagram that shows the structure of that structural element. Each UML diagram was constructed according to the provisions of the ISO/IEC 19505-1 and ISO/IEC 19505-2 standards. Major subsections that describe XML structures of the complex, but not mixed, type then are divided into two further subsections — one describing the XML attributes of the structure and the other describing the XML child elements of the structure. Within the subsections on XML attributes and XML elements, each of the data points is described in the form of an XSD snippet and a semantics description. In major subsections describing complex type XML structures in the mixed types category, there are attributes associated with the structure but no child elements — only a value. In the descriptions of those mixed structures, there are lower-level subsections for the attributes and for the value of the structure. In subsections describing XML structures of simple type, there are no subsidiary attributes or elements. Thus, no subdivision is necessary, and only an XSD snippet and a semantic description of the entire structure are provided.

Associated with this standard is an XML Schema Definition (XSD) file, named <ST2034-1a,xsd>, which provides complete descriptions of the XML data structures in machine and (almost) human-readable form. The XSD file comprises an element of this standard, and a link to it can be found at <http://www.smpte-ra.org/ns/2034-1/2017/AXF>. The Version attribute in the schema header of the XSD file associated with this version of ST 2034-1, AXF Part 1, has a value of 1.1. The XSD file includes comments that fully describe the semantics of each data point and are derived from the normative requirements in this section. The prose descriptions in this section shall express the normative semantics for each XML attribute or element; the text documentation in the XSD file is for information purposes only. This section also includes snippets derived from the XSD file that is an element of this standard. The XSD file shall be the normative expression of the XML structures defined in this standard; the XSD snippets contained herein are for informational and contextual purposes only. With respect to the existence of specific XML structures, attributes, and/or elements, the XSD file that is an element of this standard shall take precedence over this text document.

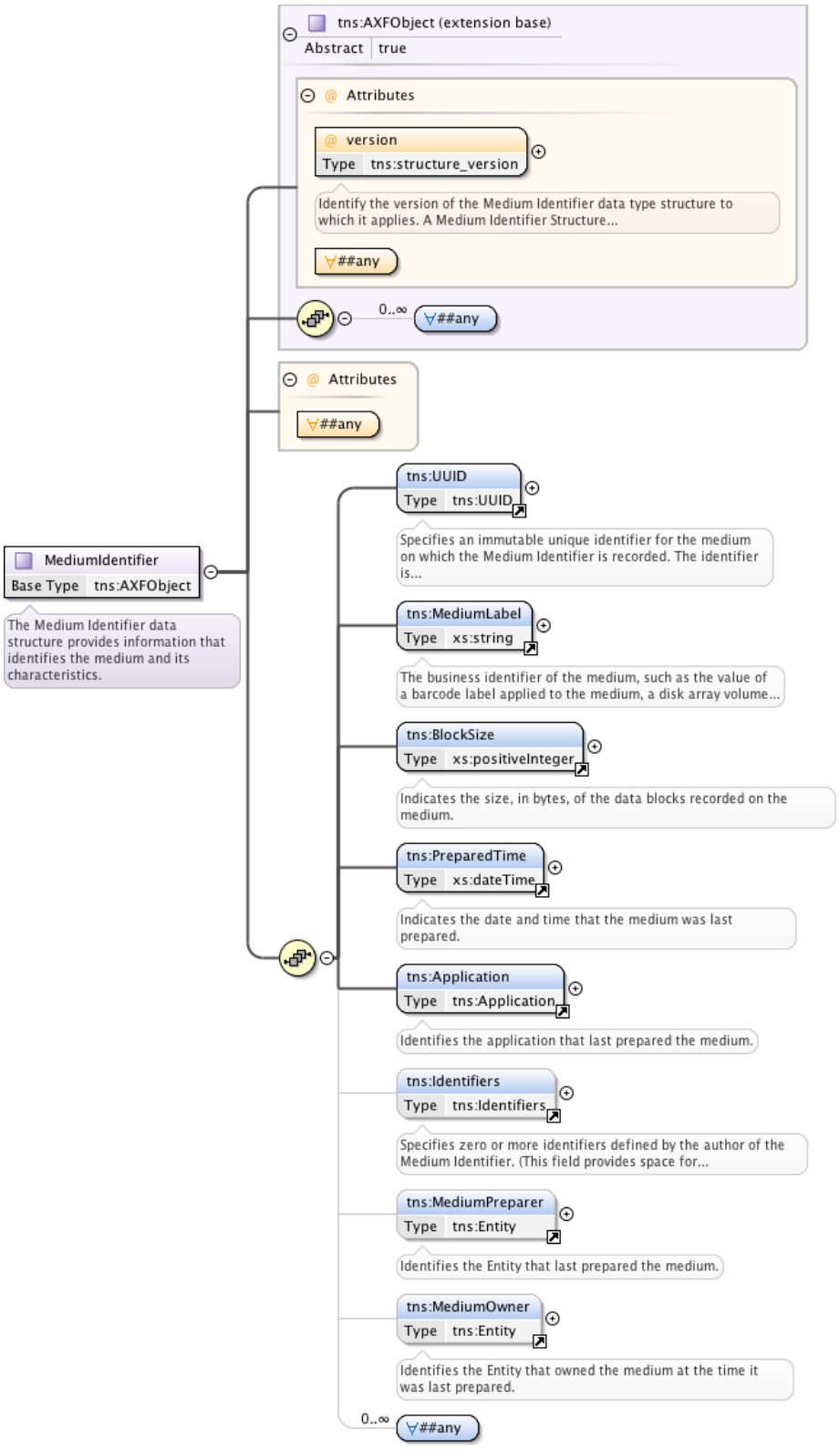
In the UML diagrams following, notes appear for each of the attributes and elements included in each structure. The notes are derived from the comments included in the XSD file. When there is more material included in a description in the XSD file than fits in the space available for a corresponding note in the UML diagram, the note ends with an ellipsis (…). In such cases, see the XSD file for the complete, non-normative documentation provided to aid understanding of the XSD file structure.

Note: Prior versions of the XSD file did not include a Version attribute in the schema header. It is suggested that AXF parsers finding AXF Objects without the Version attribute present consider the AXF Object to be constructed according to Version 1.0.

Note: The headers of XML files generated in accordance with the XSD file that is an element of this standard are required to include an XML namespace that is defined in the XSD file. It is known to the authors of this standard that some implementations of earlier versions of this standard did not generate XML files that included the required namespace.

## 10.1 AXF Medium Identifier

The AXF Medium Identifier data structure provides information that identifies a medium and its characteristics.



### 10.1.1 AXF Medium Identifier Attributes

The AXF Medium Identifier attribute is: **version**

#### 10.1.1.1 Version

```
<xs:attribute name="version" type="tns:structure_version" use="required" />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the AXF Medium Identifier in which it is found. The AXF Medium Identifier Structure Version applicable to this edition of this standard shall be 1.0.

### 10.1.2 AXF Medium Identifier Elements

The AXF Medium Identifier elements are: **UUID**, **Label**, **BlockSize**, **PreparedTime**, **Application**, **Identifier**, **MediumPreparer**, **MediumOwner**.

#### 10.1.2.1 UUID

```
<xs:element name="UUID" type="tns:UUID" />
```

The Universally Unique IDentifier (**UUID**) contained within an AXF Medium Identifier shall specify a unique identifier of the medium on which the AXF Medium Identifier is recorded. The **UUID** shall be reused for the life of the medium, even if the medium is reformatted. The **UUID** shall be generated using the methods defined in RFC 4122 or ISO/IEC 9834-8.

#### 10.1.2.2 MediumLabel

```
<xs:element name="MediumLabel" type="xs:string" />
```

The **MediumLabel** contained within an AXF Medium Identifier shall be a physical identifier of the medium, such as the value of a barcode label applied to the medium on which it is found, a disk array volume label, a solid state memory device label, or some other identifying data specific to the medium at the time of its preparation.

Consideration should be given to minimizing the length of the **MediumLabel** element to the extent practical, as its value is repeated in the **PreviousMediumLabel** and **NextMediumLabel** elements of Fragment Headers and Fragment Footers, respectively.

#### 10.1.2.3 BlockSize

```
<xs:element name="BlockSize" type="xs:positiveInteger" />
```

The **BlockSize** contained within an AXF Medium Identifier shall indicate the size, in bytes, of the data blocks recorded on the medium on which it is found.

#### 10.1.2.4 PreparedTime

```
<xs:element name="PreparedTime" type="xs:dateTime" />
```

The **PreparedTime** value contained within an AXF Medium Identifier shall indicate the date and time at which the medium on which it is found last was prepared.

#### 10.1.2.5 Application

```
<xs:element name="Application" type="tns:Application" />
```

The **Application** element contained within an AXF Medium Identifier shall identify the application that last prepared the medium on which it is found. The Application element should contain the name, version, and publisher of the relevant software application.

#### 10.1.2.6 Identifiers

```
<xs:element name="Identifiers" type="tns:Identifier" />
```

The **Identifiers** element contained within an Object Header shall comprise zero or more **Identifier** data types defined by the author of the AXF Object in which it is found.

#### 10.1.2.7 MediumPreparer

```
<xs:element name="MediumPreparer" type="tns:Entity" />
```

The **MediumPreparer** element contained within an AXF Medium Identifier shall identify the Entity that last prepared the medium on which it is found. The **MediumPreparer** element shall contain the name of the Entity, should contain the name of the operator, and may contain other information pertaining to the Entity that last prepared the medium.

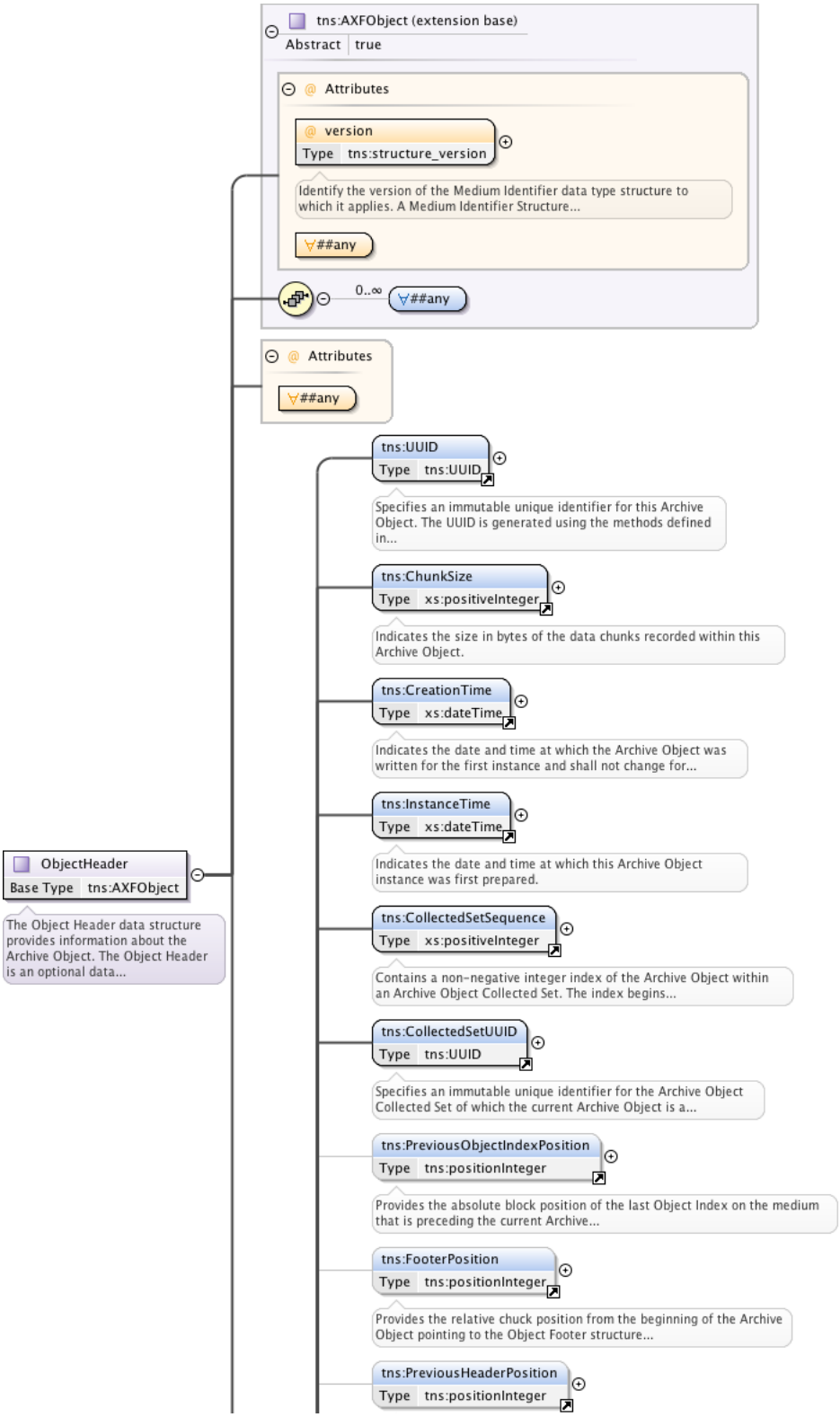
#### 10.1.2.8 MediumOwner

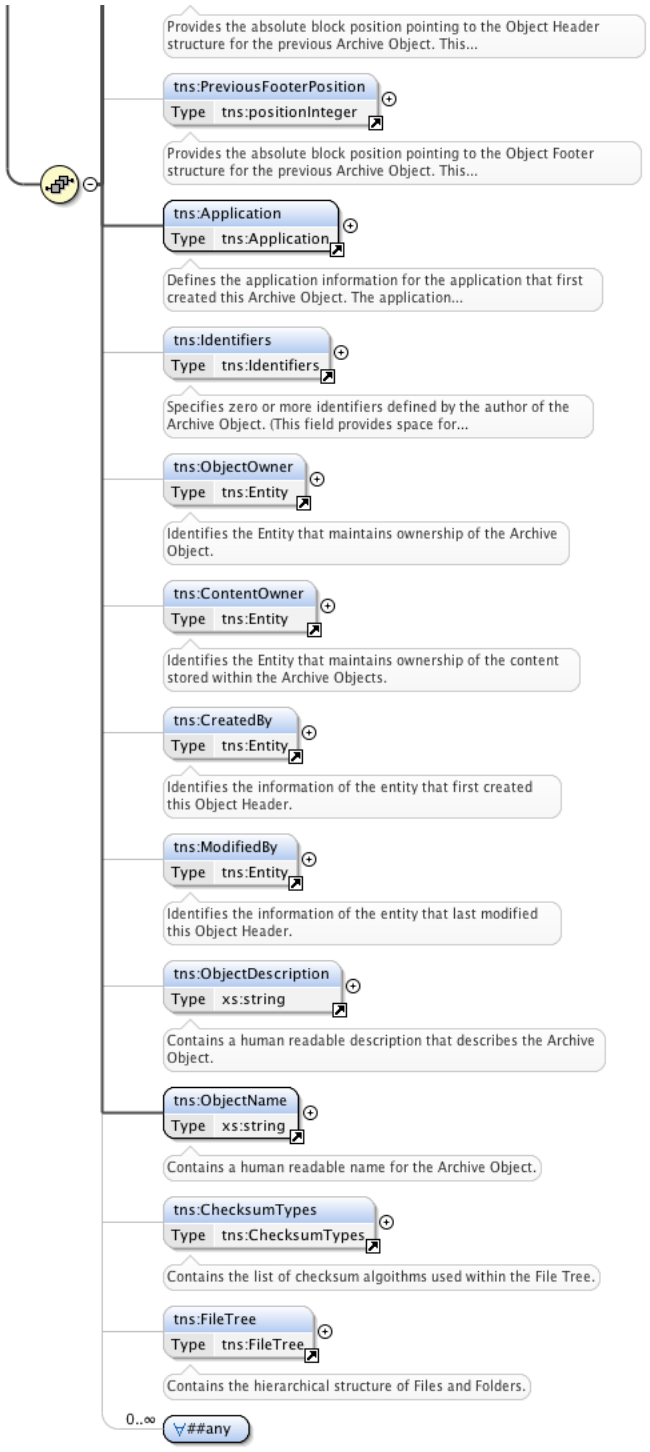
```
<xs:element name="MediumOwner" type="tns:Entity" />
```

The **MediumOwner** element contained within an AXF Medium Identifier shall identify the Entity that owned the medium on which it is found when it last was prepared. The **MediumOwner** element shall contain the name of the Entity, should contain the name of the facility, and may contain other information pertaining to the Entity that owned the medium when it last was prepared.

## 10.2 Object Header

The Object Header data structure provides information about the AXF Object in which it is found. Each AXF Object shall have an Object Header as the first data structure of the AXF Object.






### 10.2.1 Attributes

The Object Header attribute is: **version**

#### 10.2.1.1 Version

```
<xs:attribute name="version" type="tns:structure_version" use="required" />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the Object Header in which it is found. The Object Header Structure Version applicable to this edition of this standard shall be 1.1.

### 10.2.2 Elements

The Object Header elements are: **UUID**, **ChunkSize**, **InstanceTime**, **CollectedSetSequence**, **CollectedSetUUID**, **PreviousObjectIndexPosition**, **FooterPosition**, **PreviousHeaderPosition,** **PreviousFooterPosition**, **Application**, **Identifier**, **ObjectOwner**, **ContentOwner**, **CreatedBy**, **ModifiedBy**, **ObjectDescription**, **ObjectName**, **ChecksumTypes**, and **FileTree**.

#### 10.2.2.1 UUID

```
<xs:element name="UUID" type="tns:UUID" />
```

The Universally Unique IDentifier (**UUID**) contained within an Object Header shall specify a unique identifier for the AXF Object in which it is found. The identifier shall apply to the AXF Object and to any and all copies thereof that are unchanged in their content from the original. The UUID shall be generated using the methods defined in RFC 4122 or ISO/IEC 9834-8.

#### 10.2.2.2 ChunkSize

```
<xs:element name="ChunkSize" type="xs:positiveInteger" />
```

The **ChunkSize** contained within an Object Header shall indicate the size in bytes of the data Chunks recorded within the AXF Object in which it is found.

#### 10.2.2.3 CreationTime

```
<xs:element name="CreationTime" type="xs:dateTime" />
```

The CreationTime contained within an Object Header shall indicate the date and time at which the Object Header was written for the first instance of the AXF Object in which it is found. The CreationTime value shall be unchanged in all subsequent instances (copies) of the AXF Object. The element name **Creationtime** shall be interpreted as identical to **CreationTime** but shall be deprecated. For newly-created AXF Objects, the element name **CreationTime** shall be applied. When AXF Objects are spawned from previously existing AXF Objects that used the element name **Creationtime**, whether while producing a Product Object from a Collected Set or while purposely updating and/or correcting an existing AXF Object, only the term **CreationTime** shall be used. At the time that AXF Objects using the form **Creationtime** are refreshed in storage, generation of a replacement AXF Object to permit adjustment of the element name to **CreationTime** is recommended.

#### 10.2.2.4 InstanceTime

```
<xs:element name="InstanceTime" type="xs:dateTime" />
```

The **InstanceTime** contained within an Object Header shall indicate the date and time at which the particular instance (copy) of the Object Header of the AXF Object in which it is found was written. In the Object Header of the first instance of the AXF Object, the values of CreationTime and InstanceTime shall be equal.

#### 10.2.2.5 CollectedSetSequence

```
<xs:element name="CollectedSetSequence" type="xs:positiveInteger" />
```

The **CollectedSetSequence** contained within an Object Header shall contain a non-negative, integer index of the AXF Object in which it is found within a Collected Set. The index shall begin with 1 and increment by 1 for each AXF Object added to the Collected Set. The element name **CollectionSetSequence** shall be interpreted as identical to **CollectedSetSequence** but shall be deprecated. For newly-created AXF Objects, the element name **CollectedSetSequence** shall be applied. When AXF Objects are spawned from previously existing AXF Objects that used the element name **CollectionSetSequence**, whether while producing a Product Object from a Collected Set or while purposely updating and/or correcting an existing AXF Object, only the term **CollectedSetSequence** shall be used. At the time that AXF Objects using the form **CollectionSetSequence** are refreshed in storage, generation of a replacement AXF Object to permit adjustment of the element name to **CollectedSetSequence** is recommended.

#### 10.2.2.6 CollectedSetUUID

```
<xs:element name="CollectedSetIUUID" type="tns:UUID" />
```

The **CollectedSetUUID** contained within an Object Header shall specify a unique identifier for the Collected Set of which the AXF Object in which it is found is a part. The **CollectedSetUUID** value shall be set equal to the UUID of the first AXF Object in the Collected Set; i.e., the AXF Object having the lowest Collected Set Sequence Number. When an AXF Object is created that is not part of a Collected Set, its **CollectedSetUUID** value shall equal the **UUID** value of that AXF Object, and its **CollectedSetSequence** number shall be set to 1. The element name **CollectionSetUUID** shall be interpreted as identical to **CollectedSetUUID** but shall be deprecated. For newly-created AXF Objects, the element name **CollectedSetUUID** shall be applied. When AXF Objects are spawned from previously existing AXF Objects that used the element name **CollectionSetUUID**, whether while producing a Product Object from a Collected Set or while purposely updating and/or correcting an existing AXF Object, only the term **CollectedSetUUID** shall be used. At the time that AXF Objects using the form **CollectionSetUUID** are refreshed in storage, generation of a replacement AXF Object to permit adjustment of the element name to **CollectedSetUUID** is recommended.

#### 10.2.2.7 PreviousObjectIndexPosition

```
<xs:element name="PreviousObjectIndexPosition" type="tns:positionInteger" />
```

The **PreviousObjectIndexPosition** contained within an Object Header shall indicate the absolute block position of the last AXF Object Index on the medium that precedes the AXF Object in which it is found.

#### 10.2.2.8 FooterPosition

```
<xs:element name="FooterPosition" type="tns:positionInteger" />
```

The **FooterPosition** contained within an Object Header shall indicate the Chunk position of the beginning of the Object Footer relative to the beginning of the AXF Object in which it is found. The first byte of the Object Header shall be position 0.

#### 10.2.2.9 PreviousHeaderPosition

```
<xs:element name="PreviousHeaderPosition" type="tns:positionInteger" />
```

The **PreviousHeaderPosition** contained within an Object Header shall indicate the absolute block position of the Object Header structure of the preceding AXF Object on the same medium on which is stored the AXF Object in which it is found. This item is used to assist in the recovery and indexing of media and is optional, as File-System-Based Storage Media Types might not be able to provide the information necessary to its formation.

#### 10.2.2.10 PreviousFooterPosition

```
<xs:element name="PreviousFooterPosition" type="tns:positionInteger" />
```

The **PreviousFooterPosition** contained within an Object Header shall indicate the absolute block position of the Object Footer structure of the preceding AXF Object on the same medium on which is stored the AXF Object in which it is found. This item is used to assist in the recovery and indexing of media and is optional, as File-System-Based Storage Media Types might not be able to provide the information necessary to its formation.

#### 10.2.2.11 Application

```
<xs:element name="Application" type="tns:Application" />
```

The **Application** complex data type contained within an Object Header shall identify the application that first created the AXF Object in which it is found. The **Application** complex data type should contain the name, version, and publisher of the relevant software application.

#### 10.2.2.12 Identifiers

```
<xs:element name="Identifiers" type="tns:Identifier" />
```

The **Identifiers** element contained within an Object Header shall comprise zero or more **Identifier** data types defined by the author of the AXF Object in which it is found.

#### 10.2.2.13 ObjectOwner

```
<xs:element name="ObjectOwner" type="tns:Entity" />
```

The **ObjectOwner** element contained within an Object Header shall identify the Entity that held ownership of the AXF Object in which it is found when that AXF Object was created. The **ObjectOwner** element shall contain the name of the Entity, should contain the name of the facility, and may contain other information pertaining to the Entity that held ownership of that AXF Object when it was created.

#### 10.2.2.14 ContentOwner

```
<xs:element name="ContentOwner" type="tns:Entity" />
```

The **ContentOwner** element contained within an Object Header shall identify the Entity that held ownership of the content stored within the AXF Object in which it is found when that AXF Object was created. The **ContentOwner** element shall contain the name of the Entity and may contain other information pertaining to the Entity that held ownership of the content stored in that AXF Object when it was created.

#### 10.2.2.15 CreatedBy

```
<xs:element name="CreatedBy" type="tns:Entity" />
```

The **CreatedBy** element contained within an Object Header shall identify the Entity that created the AXF Object in which it is found. The **CreatedBy** element shall contain the name of the Entity, should contain the name of the facility, and may contain other information pertaining to the Entity that created that AXF Object.

#### 10.2.2.16 ModifiedBy

```
<xs:element name="ModifiedBy" type="tns:Entity" />
```

The **ModifiedBy** element contained within an Object Header shall identify the Entity that last modified the AXF Object in which it is found. The **ModifiedBy** element shall contain the name of the Entity, should contain the name of the facility, and may contain other information pertaining to the Entity that last modified the AXF Object.

#### 10.2.2.17 ObjectDescription

```
<xs:element name="ObjectDescription" type="xs:string" />
```

The **ObjectDescription** element contained within an Object Header shall contain a human-readable description of the AXF Object in which it is found.

#### 10.2.2.18 ObjectName

```
<xs:element name="ObjectName" type="xs:string" />
```

The **ObjectName** element contained within an Object Header shall contain a human-readable name for the AXF Object in which it is found.

#### 10.2.2.19 ChecksumTypes

```
<xs:element name="ChecksumTypes" type="tns:ChecksumTypes" />
```

The **ChecksumTypes** element contained within an Object Header shall contain a list of all the checksum algorithms specified within the **FileTree** and/or **FileFooter** structures of the related AXF Object.

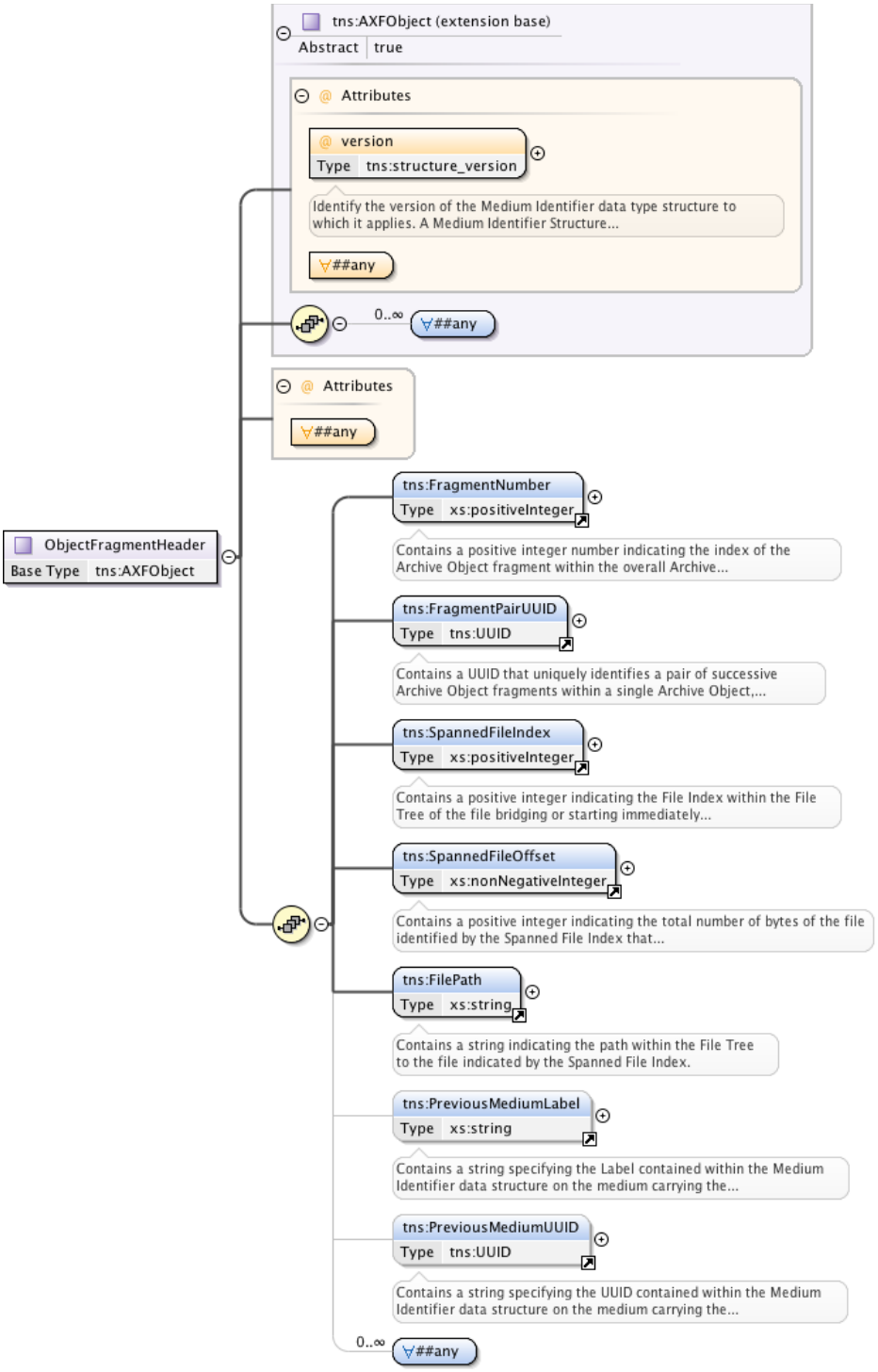
#### 10.2.2.20 FileTree

```
<xs:element name="FileTree" type="tns:FileTree" />
```

The **FileTree** element contained within an Object Header shall describe the hierarchical structure of Files and Folders stored within the AXF Object in which it is found.

## 10.3 Object Fragment Header

The Object Fragment Header contains the information necessary to connect the beginning of an AXF Object Fragment to the end of the preceding AXF Object Fragment in a Spanned Set. It should be recognized that spanning occurs on file boundaries or within files; folders cannot be spanned because they exist only in the File Tree data structure. An Object Fragment Header shall be placed at the start of an AXF Object Fragment when that fragment is not the first fragment in an AXF Object and immediately shall follow an Object Header.



### 10.3.1 Attributes

The Object Fragment Header attribute is: **version**.

#### 10.3.1.1 Version

```
<xs:attribute name=“version” type=“tns:structure_version” use=“required” />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the Object Fragment Header in which it is found. The Object Fragment Header Structure Version applicable to this edition of this standard shall be 1.0.

### 10.3.2 Elements

The Object Fragment Header elements are: **FragmentNumber**, **FragmentPairUUID**, **SpannedFileIndex**, **SpannedFileOffset**, **FilePath**, **PreviousMediumLabel**, and **PreviousMediumUUID**.

#### 10.3.2.1 FragmentNumber

```
<xs:element name="FragmentNumber" type="xs:positiveInteger" />
```

The **FragmentNumber** element contained within an Object Fragment Header shall contain a positive integer number indicating the index of the AXF Object fragment in which it is found within the overall AXF Object, starting at 1 and incrementing by 1 for each successive AXF Object Fragment.

#### 10.3.2.2 FragmentPairUUID

```
<xs:element name="FragmentPairUUID" type="tns:UUID" />
```

The **FragmentPairUUID** element contained within an Object Fragment Header shall contain a UUID that uniquely identifies a pair of successive AXF Object Fragments of a single AXF Object, in which pair the Fragment Numbers of the pair of Fragments differ by 1. The **FragmentPairUUID** shall be placed in the Object Fragment Footer of the lower-ordered Fragment and in the Object Fragment Header of the higher-ordered Fragment of the Fragment pair.

#### 10.3.2.3 SpannedFileIndex

```
<xs:element name="SpannedFileIndex" type="xs:positiveInteger" />
```

The **SpannedFileIndex** element contained within an Object Fragment Header shall contain a positive integer value indicating the File Index, within the File Tree of the AXF Object within which it is found, of the file bridging, or starting immediately following, the span; i.e., immediately following the Object Fragment Header in which the **SpannedFileIndex** is found.

#### 10.3.2.4 SpannedFileOffset

```
<xs:element name="SpannedFileOffset" type="xs:nonNegativeInteger" />
```

The **SpannedFileOffset** element contained within an Object Fragment Header shall contain a positive integer value indicating the total number of bytes of a file contained within all preceding AXF Object Fragments of the AXF Object within which the **SpannedFileOffset** element is found. The file to which the **SpannedFileOffset** element shall apply is the file identified by the **SpannedFileIndex** carried in the same Object Fragment Header as that in which the **SpannedFileOffset** is found.

#### 10.3.2.5 FilePath

```
<xs:element name="FilePath" type="xs:string" />
```

The **FilePath** element contained within an Object Fragment Header shall contain a string indicating the path within the File Tree to the file indicated by the **SpannedFileIndex** contained in the same Object Fragment Header as that in which the **FilePath** element is found, relative to the root of the AXF Object. The string contained in the **FilePath** element shall express the File Path starting at the root of the AXF Object and shall include the names of all folders on the path from the AXF Object root to and including the file indicated by the **SpannedFileIndex**. The root shall be indicated with a virgule (“/”), and all nested folder names and the name of the file indicated by the **SpannedFileIndex** shall be separated by virgules.

#### 10.3.2.6 PreviousMediumLabel

```
<xs:element name="PreviousMediumLabel" type="xs:string" />
```

The **PreviousMediumLabel** element contained within an Object Fragment Header shall contain a string specifying the Label contained within the AXF Medium Identifier data structure of the medium carrying the preceding AXF Object Fragment in the set of spanned Fragments of the AXF Object in which it is found.

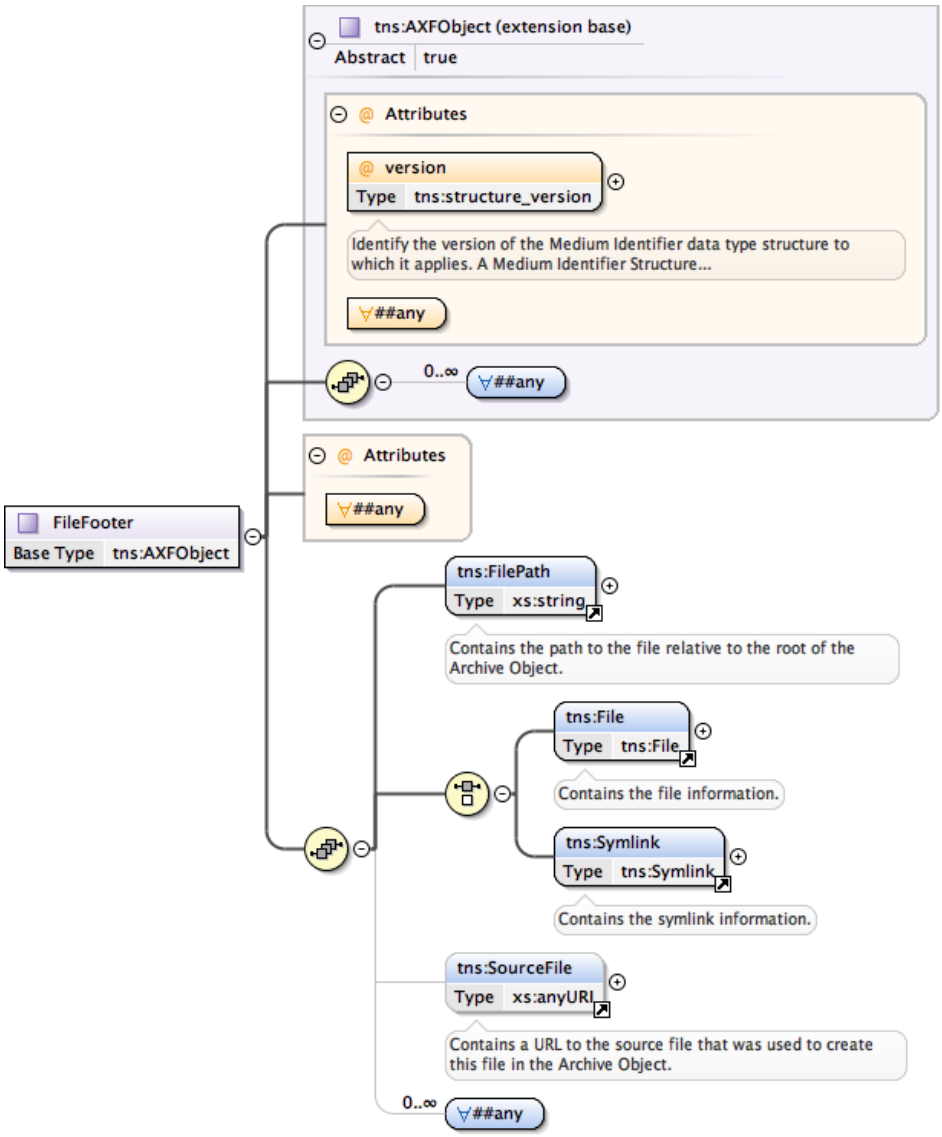
#### 10.3.2.7 PreviousMediumUUID

```
<xs:element name="PreviousMediumUUID" type="tns:UUID" />
```

The **PreviousMediumUUID** element contained within an Object Fragment Header shall contain a string specifying the UUID contained within the AXF Medium Identifier data structure of the medium carrying the preceding AXF Object Fragment in the set of spanned Fragments of the AXF Object in which it is found.

## 10.4 File Footer

The File Footer complex data type structure describes the file in the File Payload that precedes it. The File Footer is located within the File Payload section of the AXF Object in the Chunk immediately following the file that it describes.




### 10.4.1 File Footer Attributes

The File Footer attribute is: **version**.

#### 10.4.1.1 Version

```
<xs:attribute name="version" type="tns:structure_version" use="required" />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the File Footer in which it is found. The File Footer Structure Version applicable to this edition of this standard shall be 1.1.

### 10.4.2 File Footer Elements

The File Footer elements are: **FilePath**, **File**, **Symlink**, and **SourceFile**.

```
<xs:complexType name="FileFooter">
  <xs:complexContent>
    <xs:extension base="tns:AXFObject">
      <xs:sequence>
        <xs:element ref="tns:FilePath" minOccurs="1" maxOccurs="1" />
        <xs:choice minOccurs="1" maxOccurs="1">
          <xs:element ref="tns:File" />
          <xs:element ref="tns:Symlink" />
        </xs:choice>
        <xs:element ref="tns:SourceFile" minOccurs="0" maxOccurs="1" />
        <xs:any minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:anyAttribute/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

#### 10.4.2.1 FilePath

```
<xs:element name="FilePath" type="xs:string" />
```

The **FilePath** element contained within a **File Footer** shall contain the path to the file relative to the root of the AXF Object. The string contained in the **FilePath** element shall express the File Path starting at the root of the AXF Object and shall include the names of all folders on the path from the AXF Object root to and including the file followed by the **File Footer**. The root shall be indicated with a virgule (“/”), and all nested folder names and the name of the file followed by the **File Footer** in which the **FilePath** element is found shall be separated by virgules.

#### 10.4.2.2 File

```
<xs:element name="File" type="tns:File" />
```

The **File** element contained within a **File Footer** shall contain the **File** information provided by the File data type for the file indicated by the File Footer in which it is found.

#### 10.4.2.3 Symlink

```
<xs:element name="Symlink" type="tns:Symlink" />
```

The **Symlink** element contained within a **File Footer** shall contain the **Symlink** information provided by the Symlink data type for the symbolic link indicated by the File Footer in which it is found.

#### 10.4.2.4 SourceFile

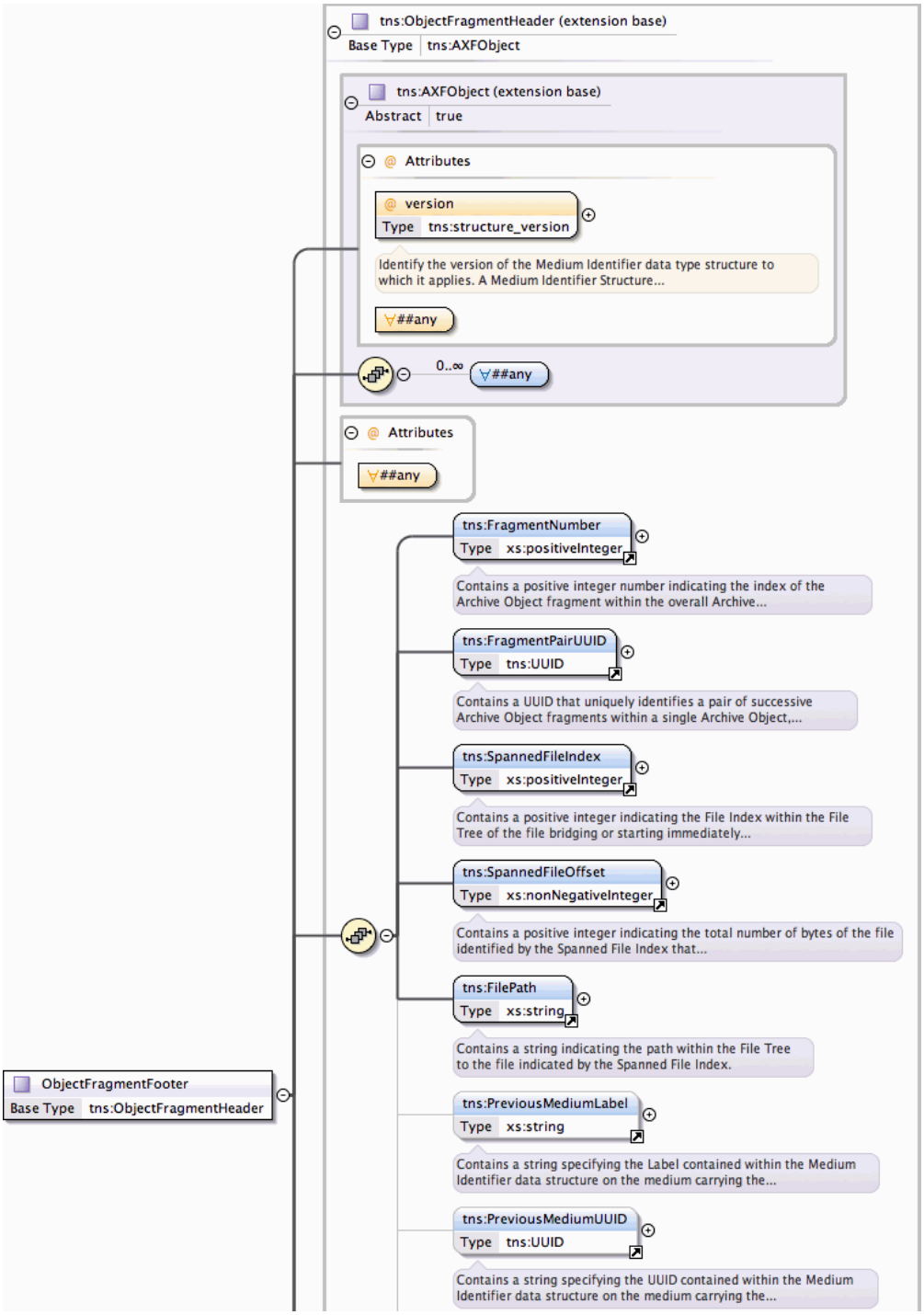
```
<xs:element name="SourceFile" type="xs:anyURI" />
```

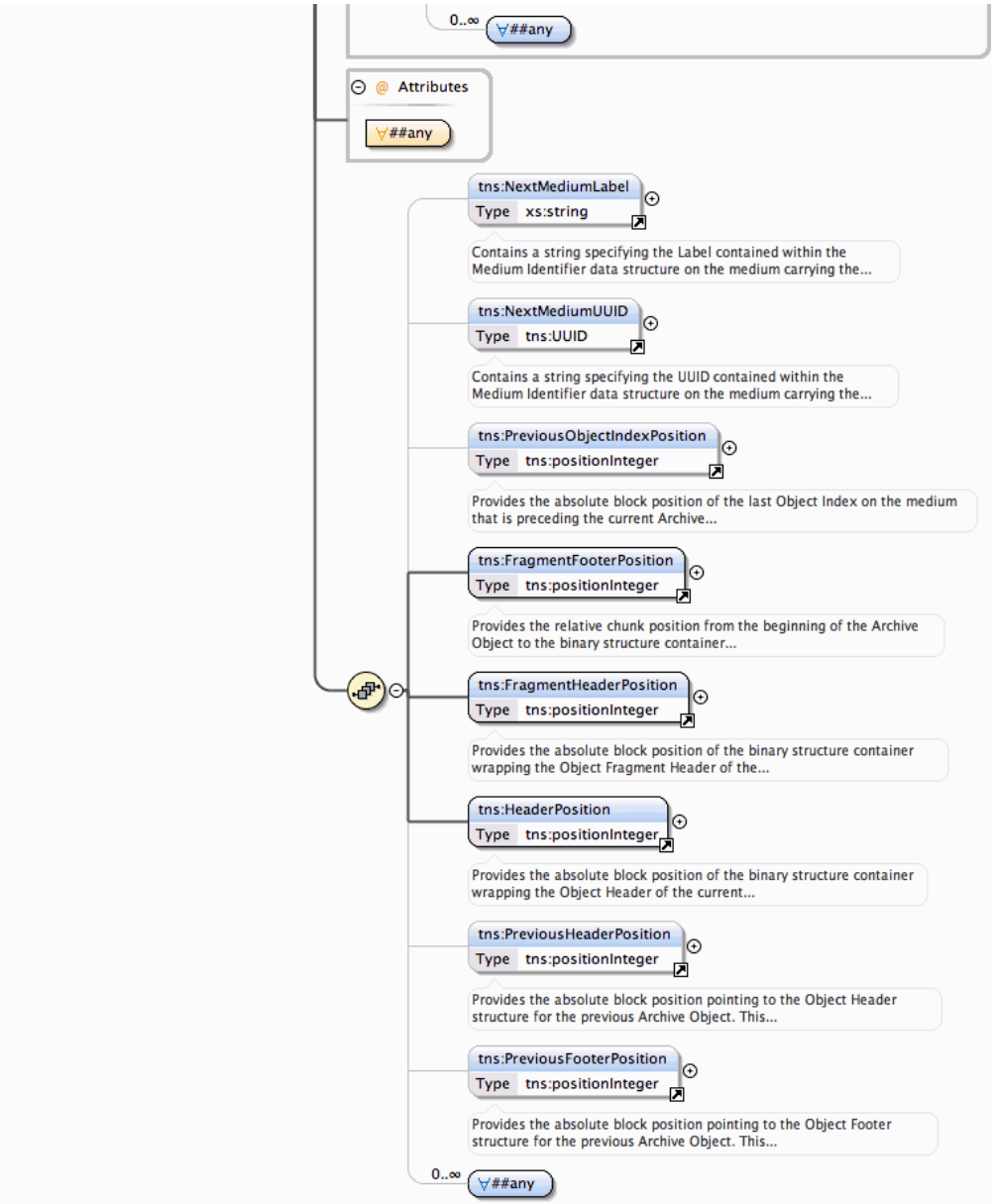
The **SourceFile** element contained within a **File Footer** shall contain a URL pointing to the source file from which the file to which the **File Footer** relates was copied into the AXF Object.

## 10.5 Object Fragment Footer

The Object Fragment Footer complex data type contains the information necessary to connect the end of an AXF Object Fragment to the beginning of the following AXF Object Fragment in a Spanned Set. An Object Fragment Footer shall substitute for an Object Footer at the end of an AXF Object Fragment when that Fragment is not the last Fragment in the AXF Object.

Note: Spanning occurs on file boundaries or within files; folders cannot be spanned because they exist only in the File Tree data structure.






### 10.5.1 Object Fragment Footer Attributes

The Object Fragment Footer attribute is: **version**.

#### 10.5.1.1 Version

```
<xs:attribute name="version" type="tns:structure_version" use="required" />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the Object Fragment Footer in which it is found. The Object Fragment Footer Structure Version applicable to this edition of this standard shall be 1.0.

### 10.5.2 Object Fragment Footer Elements

The Object Fragment Footer elements are: **FragmentNumber**, **FragmentPairUUID**, **SpannedFileIndex**, **SpannedFileOffset**, **FilePath**, **PreviousMediumLabel**, **PreviousMediumUUID**, **NextMediumLabel**, **NextMediumUUID**, **PreviousObjectIndexPosition**, **FragmentHeaderPosition**, **HeaderPosition**, **PreviousHeaderPosition**, and **PreviousFooterPosition**.

#### 10.5.2.1 FragmentNumber

```
<xs:element name="FragmentNumber" type="xs:positiveInteger" />
```

The **FragmentNumber** element contained within an Object Fragment Footer shall contain a positive integer number indicating the index of the AXF Object Fragment within the overall AXF Object, starting at 1 and incrementing by 1 for each successive AXF Object Fragment.

#### 10.5.2.2 FragmentPairUUID

```
<xs:element name="FragmentPairUUID" type="tns:UUID" />
```

The **FragmentPairUUID** element contained within an Object Fragment Footer shall contain a UUID that uniquely identifies a pair of successive AXF Object Fragments within a single AXF Object, in which pair the Fragment Numbers of the pair of Fragments differ by 1. The **FragmentPairUUID** shall be placed in the Object Fragment Footer of the lower-ordered Fragment and in the Object Fragment Header of the higher-ordered Fragment of the Fragment pair.

#### 10.5.2.3 SpannedFileIndex

```
<xs:element name="SpannedFileIndex" type="xs:positiveInteger" />
```

The **SpannedFileIndex** element contained within an Object Fragment Footer shall contain a positive integer value indicating the File Index, within the File Tree of the AXF Object within which it is found, of the file bridging, or starting immediately following, the span.

#### 10.5.2.4 SpannedFileOffset

```
<xs:element name="SpannedFileOffset" type="xs:nonNegativeInteger" />
```

The **SpannedFileOffset** element contained within an Object Fragment Footer shall contain a positive integer value indicating the total number of bytes of the file identified by the **SpannedFileIndex** that are contained within the current and all preceding AXF Object Fragments.

#### 10.5.2.5 FilePath

```
<xs:element name="FilePath" type="xs:string" />
```

The **FilePath** element contained within an Object Fragment Footer shall contain a string indicating the path within the File Tree to the file indicated by the Spanned File Index. The string contained in the **FilePath** element shall express the File Path starting at the root of the AXF Object and shall include the names of all folders on the path from the AXF Object root to and including the file indicated by the **SpannedFileIndex**. The root shall be indicated with a virgule (“/”), and all nested folder names and the name of the file indicated by the **SpannedFileIndex** shall be separated by virgules.

#### 10.5.2.6 PreviousMediumLabel

```
<xs:element name="PreviousMediumLabel" type="xs:string" />
```

The **PreviousMediumLabel** element contained within an Object Fragment Footer shall contain a string specifying the Label contained within the AXF Medium Identifier data structure on the medium carrying the preceding AXF Object fragment in the set of spanned fragments.

#### 10.5.2.7 PreviousMediumUUID

```
<xs:element name="PreviousMediumUUID" type="tns:UUID" />
```

The **PreviousMediumUUID** element contained within an Object Fragment Footer shall contain a string specifying the UUID contained within the AXF Medium Identifier data structure on the medium carrying the preceding AXF Object fragment in the set of spanned fragments.

#### 10.5.2.8 NextMediumLabel

```
<xs:element name="NextMediumLabel" type="xs:string" />
```

The **NextMediumLabel** element contained within an Object Fragment Footer shall contain a string specifying the Label contained within the AXF Medium Identifier data structure on the medium carrying the following AXF Object fragment in the set of spanned fragments.

#### 10.5.2.9 NextMediumUUID

```
<xs:element name="NextMediumUUID" type="tns:UUID" />
```

The **NextMediumUUID** element contained within an Object Fragment Footer shall contain a string specifying the UUID contained within the AXF Medium Identifier data structure on the medium carrying the following AXF Object fragment in the set of spanned fragments.

#### 10.5.2.10 PreviousObjectIndexPosition

```
<xs:element name="PreviousObjectIndexPosition" type="tns:positionInteger" />
```

The **PreviousObjectIndexPosition** element contained within an Object Fragment Footer shall provide the absolute block position of the last valid AXF Object Index on the medium preceding the current AXF Object.

```
<xs:element name="FragmentFooterPosition" type="tns:positionInteger" />
```

The **FragmentFooterPosition** element contained within an Object Fragment Footer shall provide the chunk position of the Object Fragment Footer relative to the chunk position of the Object Header preceding the current fragment of the AXF Object.

#### 10.5.2.11 FragmentHeaderPosition

```
<xs:element name="FragmentHeaderPosition" type="tns:positionInteger" />
```

The **FragmentHeaderPosition** element contained within an Object Fragment Footer shall provide the absolute block position of the Binary Structure Container wrapping the Object Fragment Header of the current fragment of the current AXF Object or of the Object Header of the current AXF Object when the current fragment is the first fragment in an AXF Object.

#### 10.5.2.12 HeaderPosition

```
<xs:element name="HeaderPosition" type="tns:positionInteger" />
```

The **HeaderPosition** element contained within an Object Fragment Footer shall provide the absolute block position of the Binary Structure Container wrapping the Object Header of the current AXF Object if the current Object Fragment is the first fragment of the AXF Object. For Object Fragments other than the first fragment of an AXF Object, the **HeaderPosition** value shall be set to -1 to indicate invalid or unknown data.

#### 10.5.2.13 PreviousHeaderPosition

```
<xs:element name="PreviousHeaderPosition" type="tns:positionInteger" />
```

The **PreviousHeaderPosition** element contained within an Object Fragment Footer shall provide the absolute block position pointing to the Object Header structure for the previous AXF Object on a Linear Medium. This information is optional, as it does not exist on File-System-Based Storage Media Types and is used only to assist in the recovery and indexing of Media.

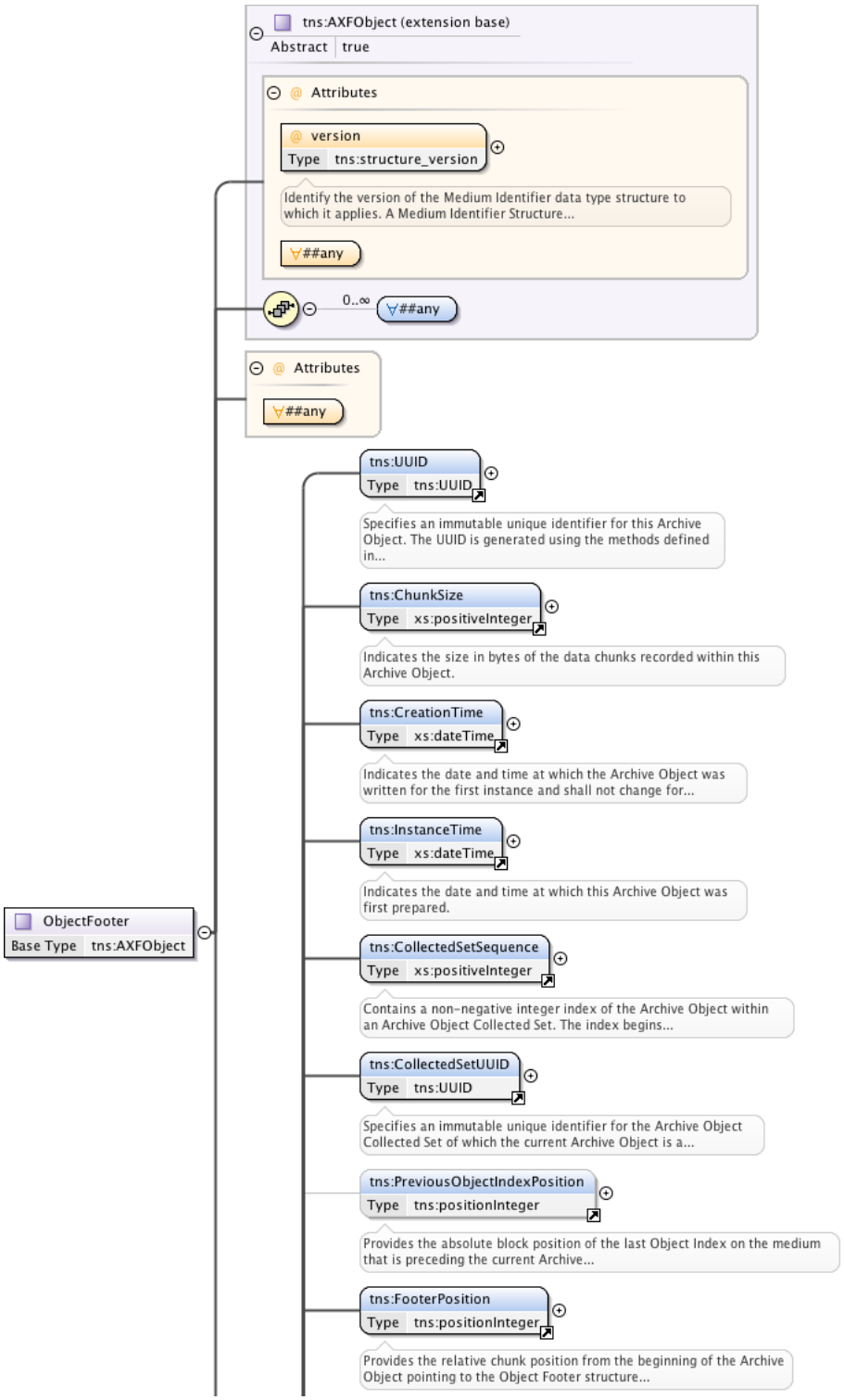
#### 10.5.2.14 PreviousFooterPosition

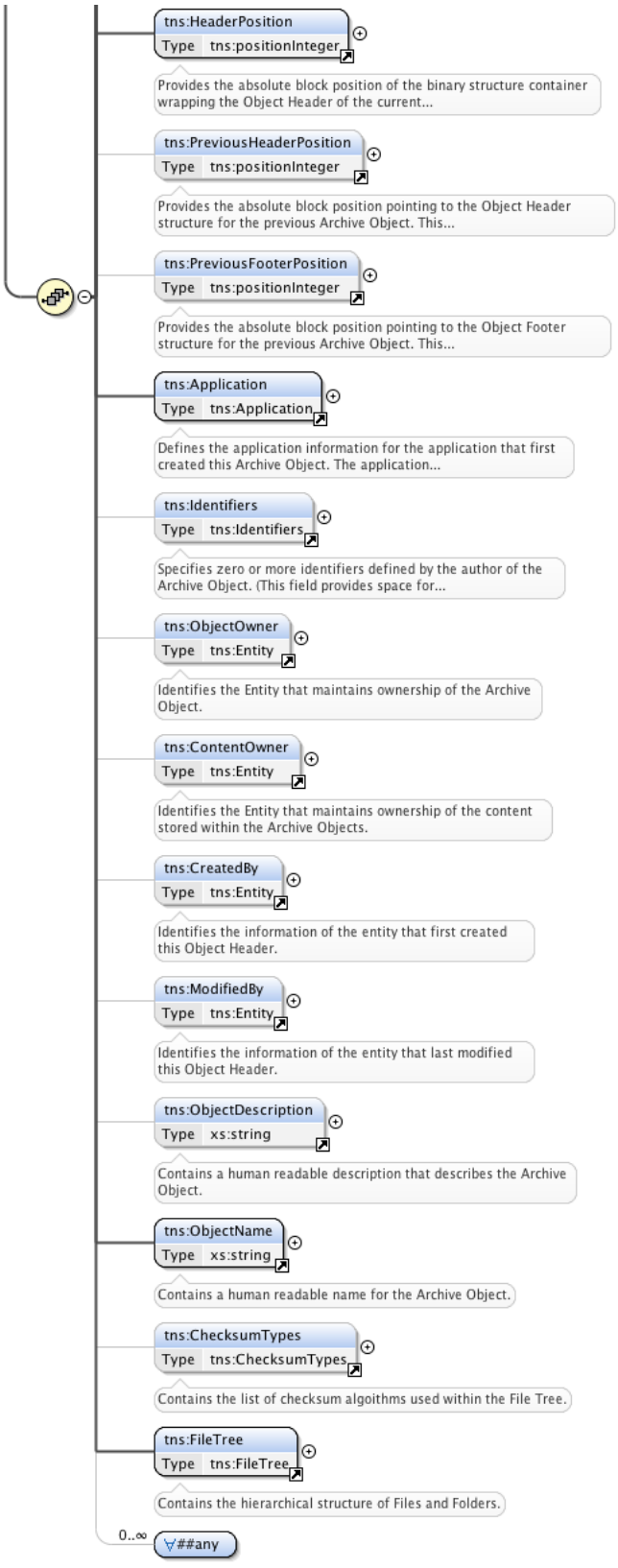
```
<xs:element name="PreviousFooterPosition" type="tns:positionInteger" />
```

The **PreviousFooterPosition** element contained within an Object Fragment Footer shall provide the absolute Block position pointing to the Object Footer structure for the previous AXF Object on a Block-Based Medium. This information is optional, as it does not exist on File-System-Based Storage Media Types and is used only to assist in the recovery and indexing of Media.

## 10.6 Object Footer

The Object Footer complex data type structure provides information about the AXF Object and the files contained therein. The Object Footer shall be the last structure of an AXF Object, closing the AXF Object, and exactly one Object Footer shall be present for each AXF Object.




tns:HeaderPosition
Type tns:positionInteger
Provides the absolute block position of the binary structure container wrapping the Object Header of the current...
tns:PreviousHeaderPosition
Type tns:positionInteger
Provides the absolute block position pointing to the Object Header structure for the previous Archive Object. This...
tns:PreviousFooterPosition
Type tns:positionInteger
Provides the absolute block position pointing to the Object Footer structure for the previous Archive Object. This...
tns:Application
Type tns:Application
Defines the application information for the application that first created this Archive Object. The application...
tns:Identifiers
Type tns:Identifiers
Specifies zero or more identifiers defined by the author of the Archive Object. (This field provides space for...
tns:ObjectOwner
Type tns:Entity
Identifies the Entity that maintains ownership of the Archive Object.
tns:ContentOwner
Type tns:Entity
Identifies the Entity that maintains ownership of the content stored within the Archive Objects.
tns:CreatedBy
Type tns:Entity
Identifies the information of the entity that first created this Object Header.
tns:ModifiedBy
Type tns:Entity
Identifies the information of the entity that last modified this Object Header.
tns:ObjectDescription
Type xs:string
Contains a human readable description that describes the Archive Object.
tns:ObjectName
Type xs:string
Contains a human readable name for the Archive Object.
tns:ChecksumTypes
Type tns:ChecksumTypes
Contains the list of checksum algoithms used within the File Tree.
tns:FileTree
Type tns:FileTree
Contains the hierarchical structure of Files and Folders.
0..∞
##any

### 10.6.1 Object Footer Attributes

The Object Footer attribute is: **version**

#### 10.6.1.1 Version

```
<xs:attribute name="version" type="tns:structure_version" use="required" />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the Object Footer in which it is found. The Object Footer Structure Version applicable to this edition of this standard shall be1.1.

### 10.6.2 Object Footer Elements

The Object Footer elements are: **UUID, ChunkSize**, **InstanceTime**, **CollectedSetSequence**, **CollectedSetIUUID**, **PreviousObjectIndexPosition**, **FooterPosition**, **HeaderPosition**, **PreviousFooterPosition**, **PreviousHeaderPosition**, **Application**, **Identifiers**, **ObjectOwner**, **ContentOwner**, **CreatedBy**, **ModifiedBy**, **ObjectDescription**, **ObjectName**, **ChecksumTypes**, and **FileTree**.

#### 10.6.2.1 UUID

```
<xs:element name="UUID" type="tns:UUID" />
```

The Universally Unique IDentifier (**UUID**) element contained within an Object Footer shall specify a unique identifier for this AXF Object. The identifier shall apply to the AXF Object and to any and all copies thereof that are unchanged in their content from the original. The **UUID** shall be generated using the methods defined in RFC 4122 or ISO/IEC 9834-8.

#### 10.6.2.2 ChunkSize

```
<xs:element name="ChunkSize" type="xs:positiveInteger" />
```

The **ChunkSize** element contained within an Object Footer shall indicate the size in bytes of the data Chunks recorded within this AXF Object.

#### 10.6.2.3 CreationTime

```
<xs:element name="CreationTime" type="xs:dateTime" />
```

The **CreationTime** element contained within an Object Footer shall indicate the date and time at which the Object Header was written for the first instance of the AXF Object. The **CreationTime** value shall be unchanged in all subsequent instances (copies) of the AXF Object. The element name **Creationtime** shall be interpreted as identical to **CreationTime** but shall be deprecated. For newly-created AXF Objects, the element name **CreationTime** shall be applied. When AXF Objects are spawned from previously existing AXF Objects that used the element name **Creationtime**, whether while producing a Product Object from a Collected Set or while purposely updating and/or correcting an existing AXF Object, only the term **CreationTime** shall be used. At the time that AXF Objects using the form **Creationtime** are refreshed in storage, generation of a replacement AXF Object to permit adjustment of the element name to **CreationTime** is recommended.

#### 10.6.2.4 InstanceTime

```
<xs:element name="InstanceTime" type="xs:dateTime" />
```

The **InstanceTime** element contained within an Object Footer shall contain the date and time value of the **InstanceTime** element of the Object Header of the AXF Object that is ended by the Object Footer in which it is found.

#### 10.6.2.5 CollectedSetSequence

```
<xs:element name="CollectedSetSequence" type="xs:positiveInteger" />
```

The **CollectedSetSequence** element contained within an Object Footer shall contain a non-negative integer index of the AXF Object within a Collected Set. The index shall begin with 1 and increment by 1 for each AXF Object added to the Collected Set.

#### 10.6.2.6 CollectedSetUUID

```
<xs:element name="CollectedSetUUID" type="tns:UUID" />
```

The **CollectedSetIUUID** element contained within an Object Footer shall specify a unique identifier for the Collected Set of which the current AXF Object is a part. The **CollectedSetUUID** shall be set equal to the UUID of the first AXF Object in the Collected Set, i.e., the AXF Object with the lowest **CollectedSetSequence** value. When an AXF Object is created that is not part of a Collected Set, its **CollectedSetUUID** shall equal the **UUID** of the AXF Object, and its Collected Set Sequence number shall be set to 1.

#### 10.6.2.7 PreviousObjectIndexPosition

```
<xs:element name="PreviousObjectIndexPosition" type="tns:positionInteger" />
```

The **PreviousObjectIndexPosition** element contained within an Object Footer shall provide the absolute Block position of the last AXF Object Index on the medium that precedes the current AXF Object.

#### 10.6.2.8 FooterPosition

```
<xs:element name="FooterPosition" type="tns:positionInteger" />
```

The **FooterPosition** element contained within an Object Footer shall provide the Chunk position, relative to the start of the AXF Object, pointing to the Object Footer structure for the current AXF Object.

#### 10.6.2.9 HeaderPosition

```
<xs:element name="HeaderPosition" type="tns:positionInteger" />
```

The **HeaderPosition** element contained within an Object Footer shall provide the absolute Block position of the Binary Structure Container wrapping the Object Header of the current AXF Object if the AXF Object is not fragmented, or of the Object Header preceding the current fragment if the fragment that the Object Footer follows is the last fragment of the AXF Object.

Note: There is no Object Footer structure following fragments of an AXF Object other than the last.

#### 10.6.2.10 PreviousHeaderPosition

```
<xs:element name="PreviousHeaderPosition" type="tns:positionInteger" />
```

The **PreviousHeaderPosition** element contained within an Object Footer shall provide the absolute Block position pointing to the Object Header structure of the previous AXF Object. This information is optional, as it does not exist on File-System-Based Storage Media Types and is used only to assist in the recovery and indexing of Media.

#### 10.6.2.11 PreviousFooterPosition

```
<xs:element name="PreviousFooterPosition" type="tns:positionInteger" />
```

The **PreviousFooterPosition** element contained within an Object Footer shall provide the absolute block position pointing to the Object Footer structure for the previous AXF Object. This information is optional, as it does not exist on File-System-Based Storage Media Types and is used only to assist in the recovery and indexing of Media.

#### 10.6.2.12 Application

```
<xs:element name="Application" type="tns:Application" />
```

The **Application** element contained within an Object Footer shall define the application information for the application that first created this AXF Object. The application information should be the name and version of the software application that first prepared the medium.

#### 10.6.2.13 Identifiers

```
<xs:element name="Identifiers" type="tns:Identifiers" />
```

The **Identifiers** element contained within an Object Footer shall specify zero or more identifiers defined by the author of the AXF Object. The Identifiers can be standardized identifiers (e.g., UMID or EIDR) or can be user-defined identifiers (e.g., Tape Number or Database Primary Key).

#### 10.6.2.14 ObjectOwner

```
<xs:element name="ObjectOwner" type="tns:Entity" />
```

The **ObjectOwner** element contained within an Object Footer shall identify the Entity that held ownership of the AXF Object when it was created.

#### 10.6.2.15 ContentOwner

```
<xs:element name="ContentOwner" type="tns:Entity" />
```

The **ContentOwner** element contained within an Object Footer shall identify the Entity that held ownership of the content stored within the AXF Object when it was created.

#### 10.6.2.16 CreatedBy

```
<xs:element name="CreatedBy" type="tns:Entity" />
```

The **CreatedBy** element contained within an Object Footer shall provide information identifying the entity that first created the Object Header in which it is found.

#### 10.6.2.17 ModifiedBy

```
<xs:element name="ModifiedBy" type="tns:Entity" />
```

The **ModifiedBy** element contained within an Object Footer shall provide information identifying the entity that last modified the Object Header in which it is found.

#### 10.6.2.18 ObjectDescription

```
<xs:element name="ObjectDescription" type="xs:string" />
```

The **ObjectDescription** element contained within an Object Footer shall contain a human-readable description of the AXF Object.

#### 10.6.2.19 ObjectName

```
<xs:element name="ObjectName" type="xs:string" />
```

The **ObjectName** element contained within an Object Footer shall contain a human-readable name for the AXF Object.

#### 10.6.2.20 ChecksumTypes

```
<xs:element name="ChecksumTypes" type="tns:ChecksumTypes" />
```

The **ChecksumTypes** element contained within an Object Footer shall contain a list of all the checksum algorithms specified within the **FileTree** and/or **FileFooter** structures of the related AXF Object.

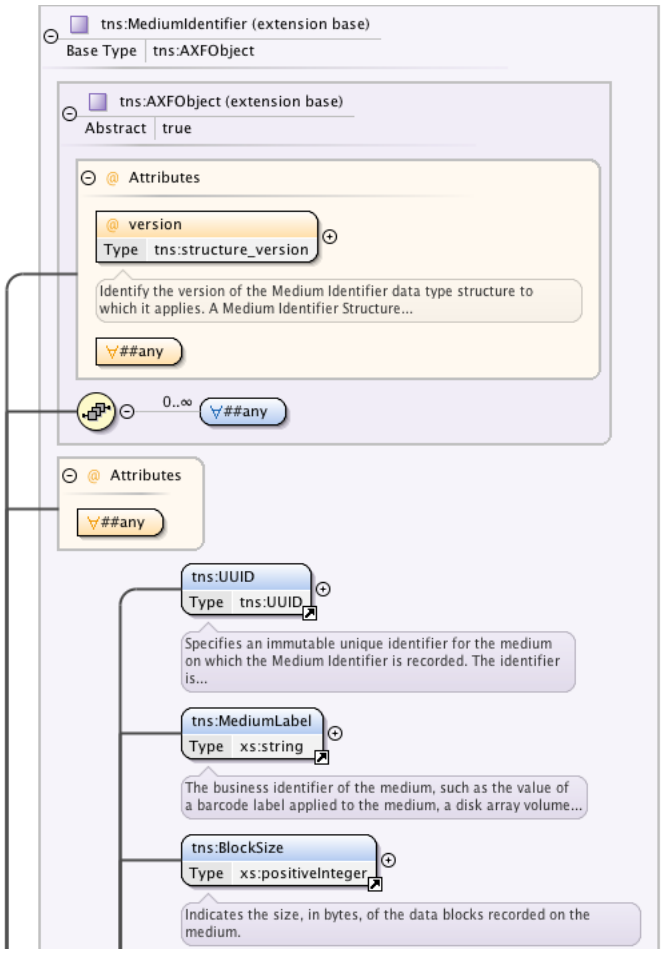
#### 10.6.2.21 FileTree

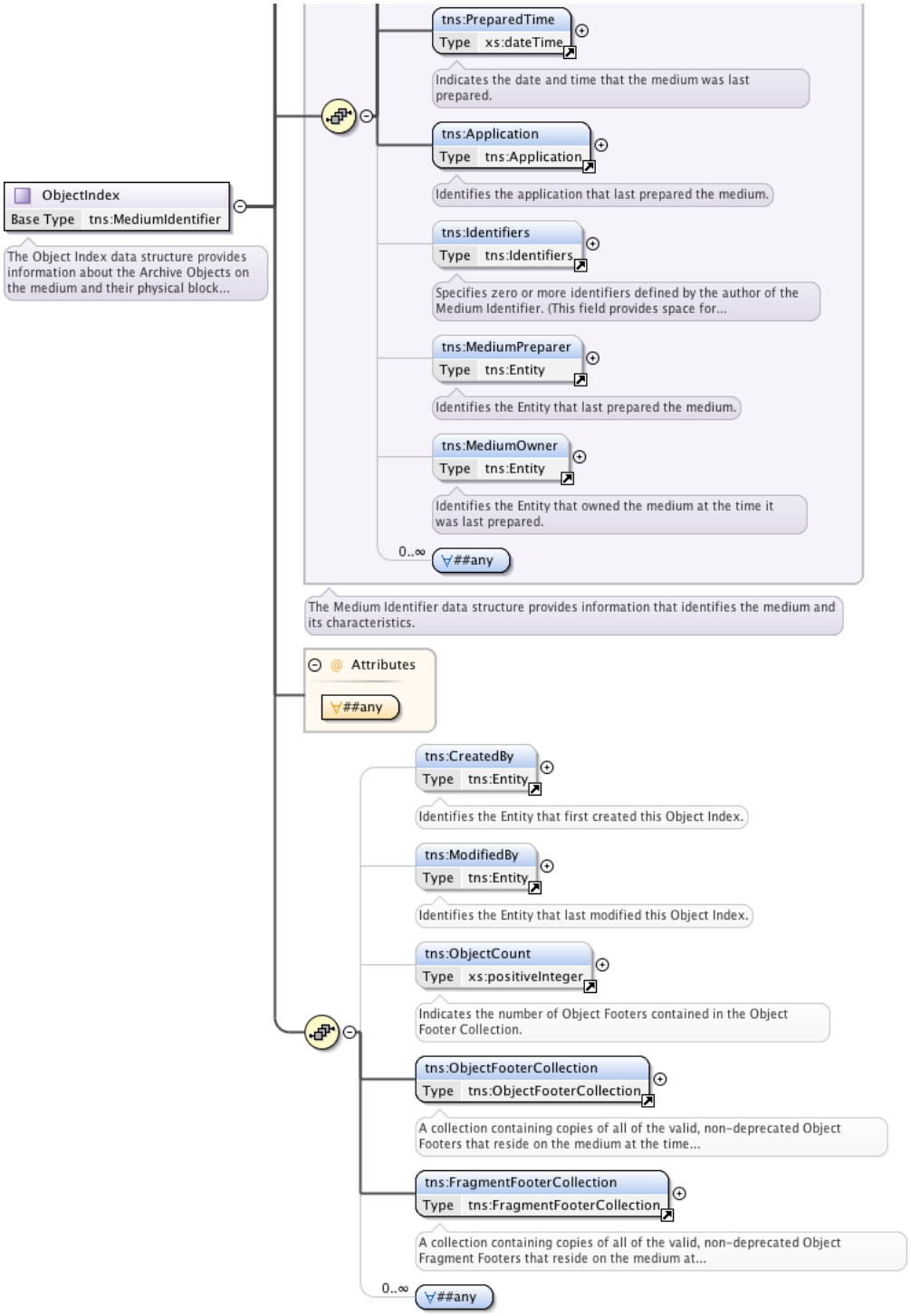
```
<xs:element name="FileTree" type="tns:FileTree" />
```

The **FileTree** element contained within an Object Footer shall contain the hierarchy of Files and Folders contained within the AXF Object.

## 10.7 AXF Object Index

The AXF Object Index complex data type structure provides information about the AXF Objects on the medium and their physical Block locations. The AXF Object Index shall contain a copy of the Object Footer from each AXF Object preceding the AXF Object Index on the medium.






### 10.7.1 AXF Object Index Attributes

The AXF Object Index attribute is: **version**

#### 10.7.1.1 Version

```
<xs:attribute name="version" type="tns:structure_version" use="required" />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the AXF Object Index in which it is found. The AXF Object Index Structure Version applicable to this edition of this standard shall be 1.1.

### 10.7.2 AXF Object Index Elements

The AXF Object Index elements are: **UUID, MediumLabel**, **BlockSize**, **PreparedTime**, **Application**, **Identifiers**, **MediumPreparer**, **MediumOwner**, **CreatedBy**, **ModifiedBy**, **ObjectCount**, **ObjectFooterSet**, **FragmentFooterSet**.

#### 10.7.2.1 UUID

```
<xs:element name="UUID" type="tns:UUID" />
```

The Universally Unique IDentifier (**UUID**) element contained within an AXF Object Index shall specify a unique identifier for the Medium on which the AXF Object Index is found. The identifier shall be reused for the life of the Medium, even if the Medium is reformatted. The UUID shall be generated using the methods defined in RFC 4122 or ISO/IEC 9834-8.

#### 10.7.2.2 MediumLabel

```
<xs:element name="Label" type="xs:string" />
```

The **MediumLabel** element contained within an AXF Object Index shall be the business identifier for the medium, such as the value of the barcode label or some other medium identifying data field specific to the Medium and the particular instantiation of the Medium.

#### 10.7.2.3 BlockSize

```
<xs:element name="BlockSize" type="xs:positiveInteger" />
```

The **BlockSize** element contained within an AXF Object Index shall indicate the size, in bytes, of the Blocks recorded on the medium.

#### 10.7.2.4 PreparedTime

```
<xs:element name="PreparedTime" type="xs:dateTime" />
```

The **PreparedTime** element contained within an AXF Object Index shall indicate the date and time at which the Medium on which it is found first was prepared.

#### 10.7.2.5 Application

```
<xs:element name="Application" type="tns:Application" />
```

The **Application** element contained within an AXF Object Index shall provide information identifying the application that first prepared the Medium on which it is found. The application information should be the name and version of the software application that first prepared the Medium.

#### 10.7.2.6 Identifiers

```
<xs:element name="Identifiers" type="tns:Identifiers" />
```

The **Identifiers** element contained within an Object Index shall specify zero or more identifiers defined by the

author of the AXF Object. The Identifiers can be standardized identifiers (e.g., UMID or EIDR) or can be user-defined identifiers (e.g., Tape Number or Database Primary Key).

#### 10.7.2.7 MediumPreparer

```
<xs:element name="MediumPreparer" type="tns:Entity" />
```

The **MediumPreparer** element contained within an AXF Object Index shall identify the Entity that last prepared the Medium on which it is found.

#### 10.7.2.8 MediumOwner

```
<xs:element name="MediumOwner" type="tns:Entity" />
```

The **MediumOwner** element contained within an AXF Object Index shall identify the Entity that owned the Medium at the time at which it last was prepared.

#### 10.7.2.9 CreatedBy

```
<xs:element name="CreatedBy" type="tns:Entity" />
```

The **CreatedBy** element contained within an AXF Object Index shall identify the Entity that first created the AXF Object Index in which it is found.

#### 10.7.2.10 ModifiedBy

```
<xs:element name="ModifiedBy" type="tns:Entity" />
```

The **ModifiedBy** element contained within an AXF Object Index shall identify the Entity that last modified the AXF Object Index in which it is found.

#### 10.7.2.11 ObjectCount

```
<xs:element name="ObjectCount" type="xs:positiveInteger" />
```

The **ObjectCount** element contained within an AXF Object Index shall indicate the number of Object Footers, copies of which are contained in the Object Footer Collection.

Note: Only the final fragments in Spanned Sets are included in Object Counts, as only final fragments are followed by Object Footers.

#### 10.7.2.12 ObjectFooterCollection

```
<xs:complexType name="ObjectFooterCollection">
  <xs:sequence>
    <xs:element name="ObjectFooter" type="tns:ObjectFooter" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:element name="ObjectFooterCollection" type="tns:ObjectFooterCollection" />
```

The **ObjectFooterCollection** element contained within an AXF Object Index shall be a collection of one copy of each of the valid, non-deprecated Object Footers that resided on the Medium at the time the AXF Object Index was created.

Note: A copy of the footer of the final fragment of a spanned set is not included in the FragmentFooterCollection, but rather in the ObjectFooterCollection, as it concludes with an Object Footer, not a Fragment Footer.

#### 10.7.2.13 FragmentFooterCollection

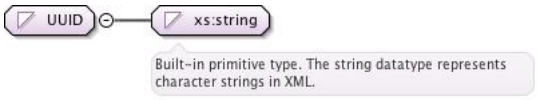
```
<xs:complexType name="FragmentFooterCollection">
  <xs:sequence>
    <xs:element name="FragmentFooter" type="tns:ObjectFragmentFooter" minOccurs="0"
                maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:element name="FragmentFooterCollection" type="tns:FragmentFooterCollection" />
```

The **FragmentFooterCollection** element contained within an AXF Object Index shall be a collection of one copy of each of the valid, non-deprecated Object Fragment Footers that resided on the Medium at the time the AXF Object Index was created.

Note: A copy of the footer of the final fragment of a spanned set is not included in the FragmentFooterCollection, but rather in the ObjectFooterCollection, as it concludes with an Object Footer, not a Fragment Footer.

### 10.8 UUID

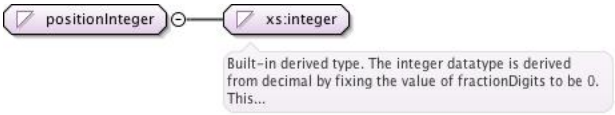
The Universal Unique IDentifier (**UUID**) structure shall define the simple data type that represents RFC 4122 or ISO/IEC 9834-8-compliant Universal Unique IDentifiers (UUIDs).




```
<xs:simpleType name="UUID">
  <xs:restriction base="xs:string">
    <xs:pattern value="[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}"/>
  </xs:restriction>
</xs:simpleType>
```

### 10.9 PositionInteger

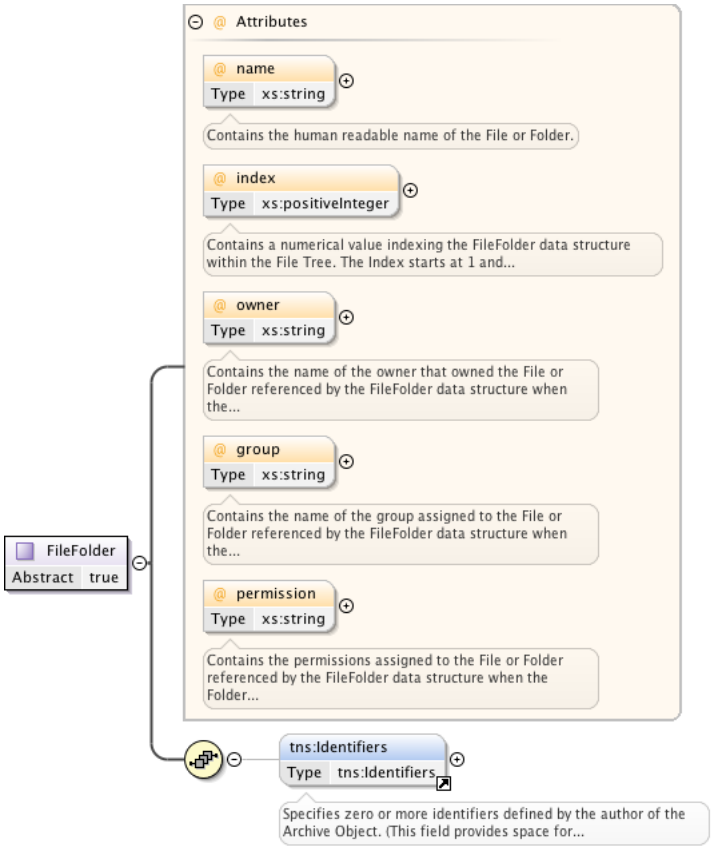
The **PositionInteger** simple data type shall be an extension of the XML integer type that ranges from -1 to positive infinity. The PositionInteger is used to represent absolute or relative positions, with a value of -1 indicating the absence of, or invalid, position information.




```
<xs:simpleType name="positionInteger" id="positionInteger">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="-1" />
  </xs:restriction>
</xs:simpleType>
```

## 10.10 FileFolder

The **FileFolder** complex data type shall define the base attributes shared by the File and the Folder data structures.




```
<xs:complexType name="FileFolder" abstract="true">
  <xs:sequence>
    <xs:element ref="tns:Identifiers" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" />
  <xs:attribute name="index" type="xs:positiveInteger" />
  <xs:attribute name="owner" type="xs:string" use="optional" />
  <xs:attribute name="group" type="xs:string" use="optional" />
  <xs:attribute name="permission" type="xs:string" use="optional" />
</xs:complexType>
```

### 10.10.1 FileFolder Attributes

The FileFolder attributes are: **name**, **index**, **owner**, **group**, and **permission**.

#### 10.10.1.1 name

The **name** attribute of the **FileFolder** data structure shall contain the human-readable name of a File or Folder.

#### 10.10.1.2 index

The **index** attribute of a **FileFolder** data structure shall contain a numerical value indexing the File Folder in which it is found and its files within its File Tree. The Index shall start at 1 and increment by 1 for each entry in the File Tree. Starting at the root, each path extending from a node shall be fully numbered (including both Folders and Files) in sequence, before proceeding to the next path extending from the same node. At any node, Folders shall be numbered at lower values than files.

#### 10.10.1.3 owner

The **owner** attribute of a **FileFolder** data structure shall contain the name of the entity that owned the File or Folder referenced by the **FileFolder** data structure when the **FileFolder** data structure was created.

#### 10.10.1.4 group

The **group** attribute of a **FileFolder** data structure shall contain the name of the group assigned to the File or Folder referenced by the **FileFolder** data structure when the **FileFolder** data structure was created.

#### 10.10.1.5 permission

The **permission** attribute of a **FileFolder** data structure shall contain the permissions assigned to the File or Folder referenced by the **FileFolder** data structure when the Folder data structure was created.

### 10.10.2 FileFolder Elements

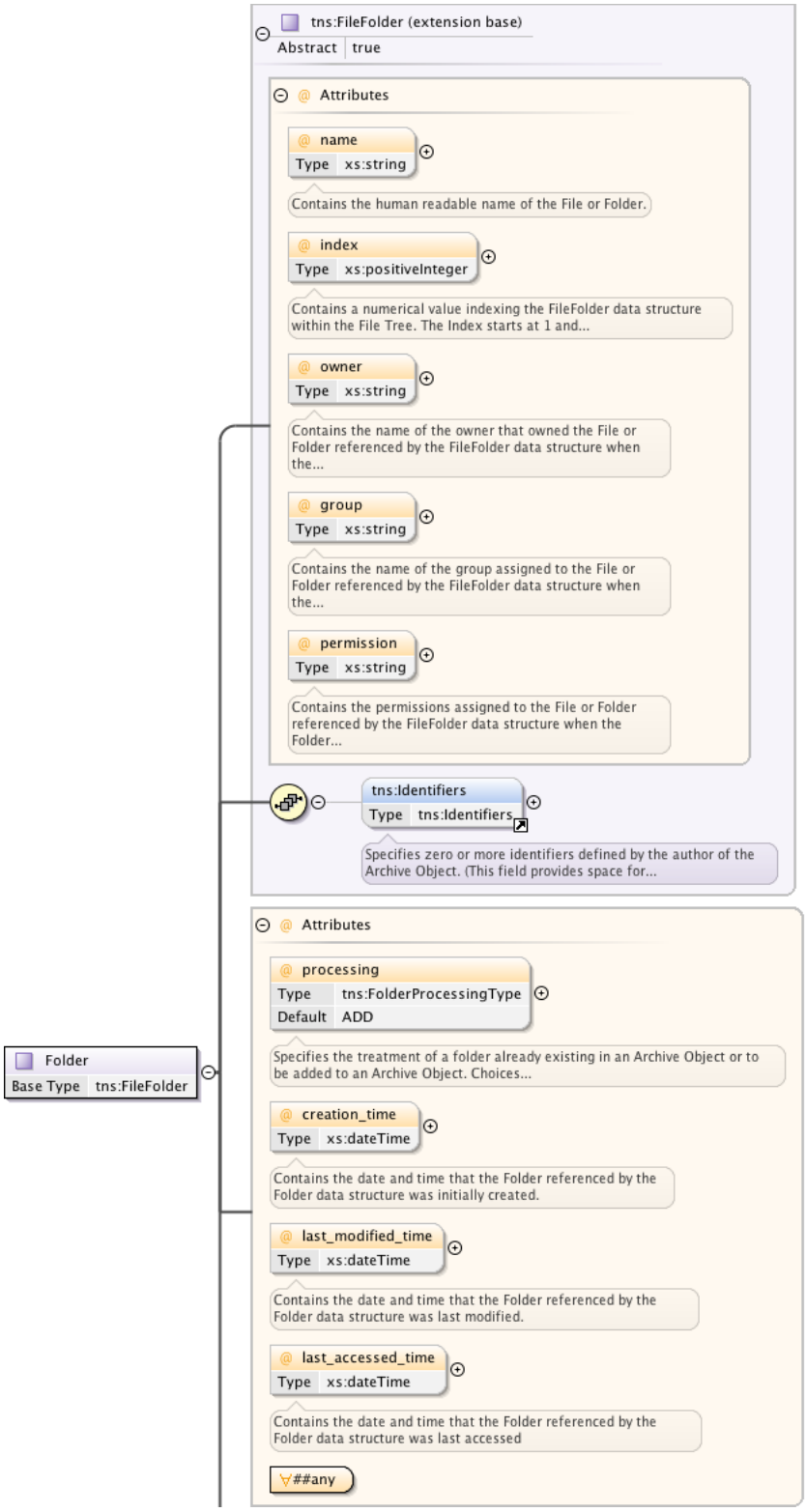
The FileFolder element is: **Identifier**

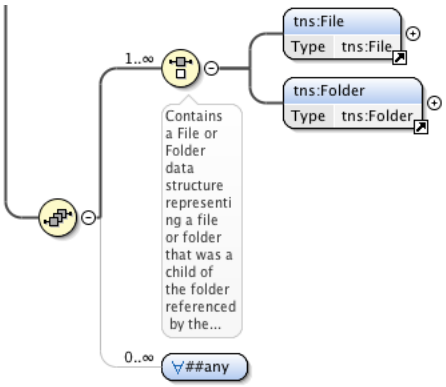
#### 10.10.2.1 Identifiers

The **Identifiers** element contained within a **FileFolder** complex data type shall specify zero or more identifiers defined by the author of the AXF Object. The Identifiers can be standardized identifiers (e.g., UMID or EIDR) or can be user-defined identifiers (e.g., Tape Number or Database Primary Key).

## 10.11 Folder

The fields contained within the **Folder** complex data type shall apply to entities referenced by higher level data structures into which the **Folder** data type is incorporated.






```
<xs:complexType name="Folder">
  <xs:complexContent>
    <xs:extension base="tns:FileFolder">
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element ref="tns:File" />
        <xs:element ref="tns:Folder" />
      </xs:choice>
      <xs:attribute name="processing" type="tns:FolderProcessingType" default="ADD" />
      <xs:attribute name="creation_time" type="xs:dateTime" use="optional" />
      <xs:attribute name="last_modified_time" type="xs:dateTime" use="optional" />
      <xs:attribute name="last_accessed_time" type="xs:dateTime" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

### 10.11.1 Folder Attributes

The Folder attributes are: **processing**, **creation_time**, **last_modified_time**, and **last_accessed_time**.

#### 10.11.1.1 processing

The **processing** attribute of a Folder data structure shall specify the treatment of a folder already existing in an AXF Object or to be added to an AXF Object. Choices are ADD or DELETE. If the attribute is not present, the default treatment shall be ADD.

#### 10.11.1.2 creation_time

The **creation_time** attribute of a Folder data structure shall contain the date and time at which the folder referenced by the Folder data structure initially was created.

#### 10.11.1.3 last_modified_time

The **last_modified_time** attribute of a Folder data structure shall contain the date and time at which the folder referenced by the Folder data structure last was modified.

#### 10.11.1.4 last_accessed_time

The **last_accessed_time** attribute of a Folder data structure shall contain the date and time at which the folder referenced by the Folder data structure last was accessed.

### 10.11.2 Folder Elements

The Folder elements are: **File** and **Folder**

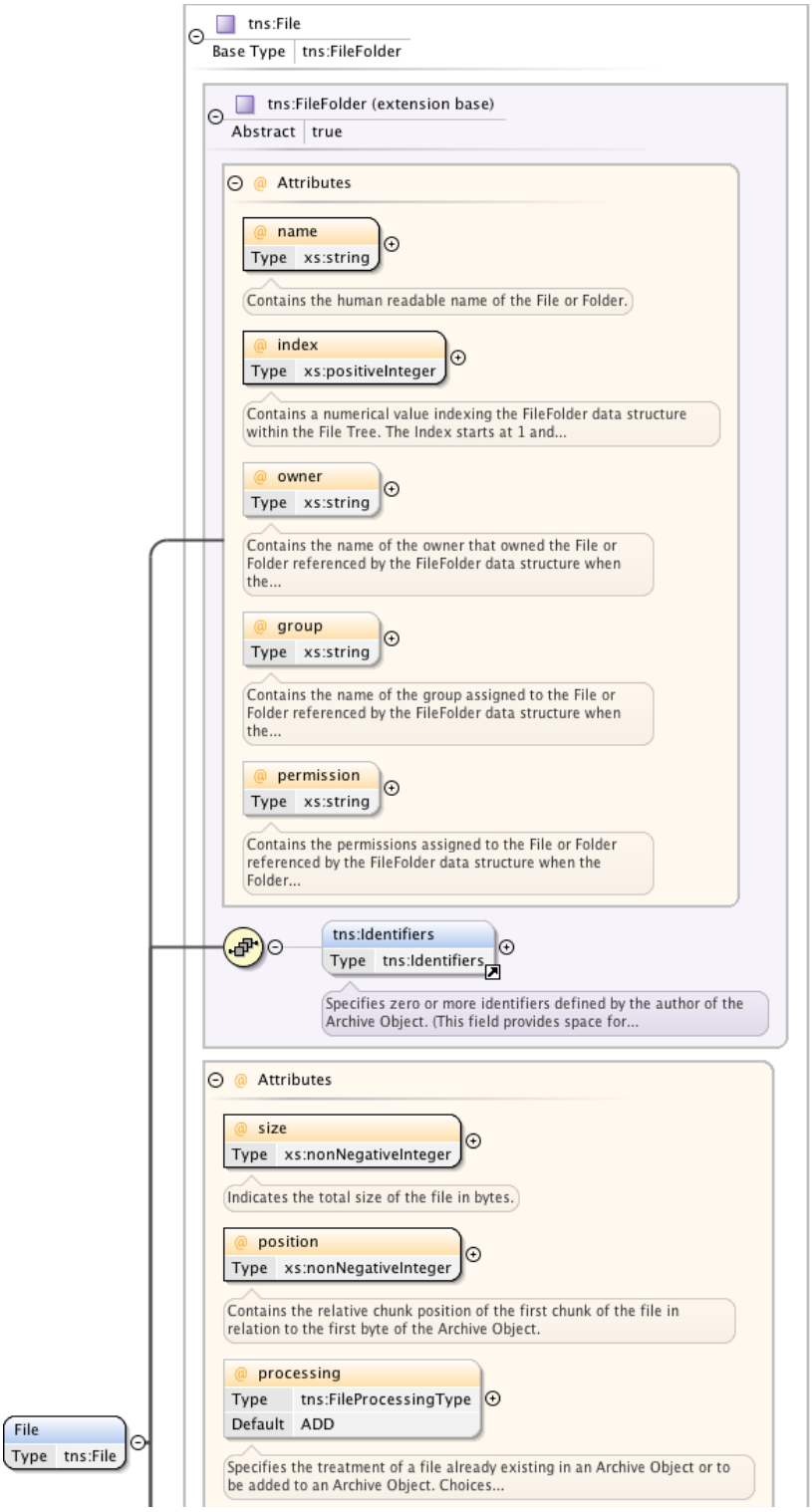
#### 10.11.2.1 File or Folder

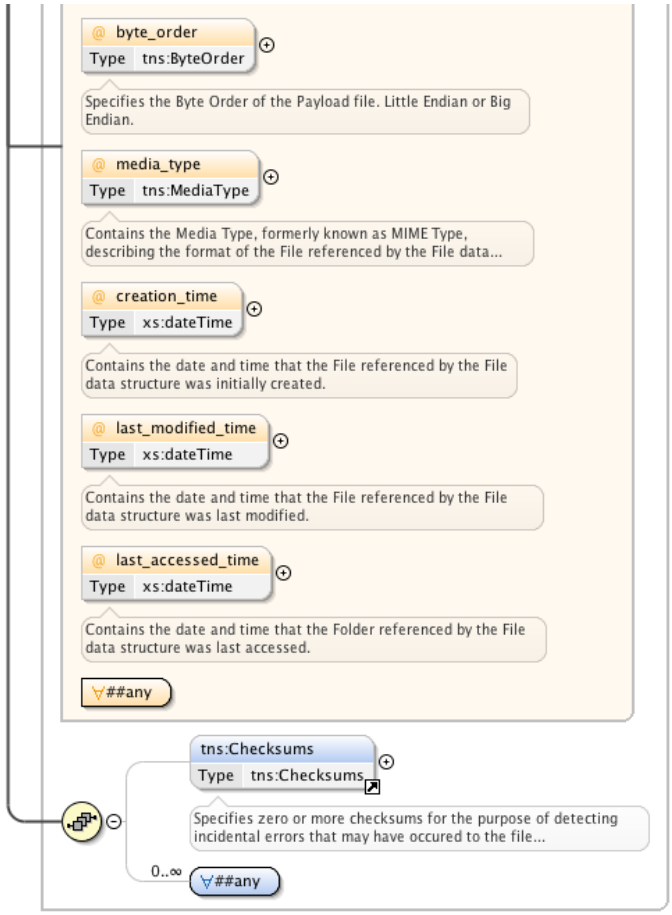
```
<xs:choice minOccurs="1" maxOccurs="unbounded">
   <xs:element ref="tns:File" />
   <xs:element ref="tns:Folder" />
</xs:choice>
```

The **File** and **Folder** elements within a Folder data type structure shall be File or Folder data type structures themselves, representing files or folders that are children one level lower in the folder hierarchy than the subject folder in which they are contained.

## 10.12 File

The fields contained within the **File** complex data type shall apply to entities referenced by higher-level data structures into which the **File** data type is incorporated.






```
<xs:complexType name="File">
  <xs:complexContent>
    <xs:extension base="tns:FileFolder">
      <xs:sequence>
        <xs:element ref="tns:Checksums" minOccurs="0" maxOccurs="1" />
      </xs:sequence>
      <xs:attribute name="size" type="xs:nonNegativeInteger" use="required" />
      <xs:attribute name="position" type="xs:nonNegativeInteger" use="required" />
      <xs:attribute name="processing" type="tns:FileProcessingType" default="ADD" />
      <xs:attribute name="byte_order" type="xs:ByteOrder" use="optional" />
      <xs:attribute name="media_type" type="tns:MediaType" use="optional" />
      <xs:attribute name="creation_time" type="xs:dateTime" use="optional" />
      <xs:attribute name="last_modified_time" type="xs:dateTime" use="optional" />
      <xs:attribute name="last_accessed_time" type="xs:dateTime" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

### 10.12.1 File Attributes

The **File** attributes are: **size**, **position**, **processing, byte_order, media_type, creation_time, last_modified_time, and last_accessed_time.**

#### 10.12.1.1 size

The **size** attribute of a **File** data structure shall indicate the total size of the file in bytes.

#### 10.12.1.2 position

The **position** attribute of a **File** data structure shall be a positive integer value identifying the relative Chunk position of the first Chunk of the file in relation to the Chunk containing the first byte of the AXF Object.

#### 10.12.1.3 processing

The **processing** attribute of a **File** data structure shall specify the treatment of a file already existing in an AXF Object or to be added to an AXF Object. Choices are ADD, REPLACE, and DELETE. If the attribute is not present, the default treatment shall be ADD. (See Section 9.3.)

#### 10.12.1.4 byte_order

The **byte_order** attribute of a **File** data structure shall indicate the byte ordering of the associated Payload File. “LE” shall indicate Little Endian; “BE” shall indicate Big Endian; “BOM” shall indicate that a Byte Order Mark is contained within the associated Payload File. The **byte_order** attribute shall be omitted either when the specific byte order is unknown or when it is unknown whether the associated file contains its own Byte Order Mark.

#### 10.12.1.5 media_type

The **media_type** attribute of a **File** data structure shall indicate the Media Type of the associated file, in accordance with RFC 6838.

#### 10.12.1.6 creation_time

The **creation_time** attribute of a **File** data structure shall contain the date and time at which the File referenced by the **File** data structure initially was created.

#### 10.12.1.7 last_modified_time

The **last_modified_time** attribute of a **File** data structure shall contain the date and time at which the file referenced by the **File** data structure last was modified.

#### 10.12.1.8 last_accessed_time

The **last_accessed_time** attribute of a **File** data structure shall contain the date and time at which the file referenced by the **File** data structure last was accessed.

### 10.12.2 File Elements

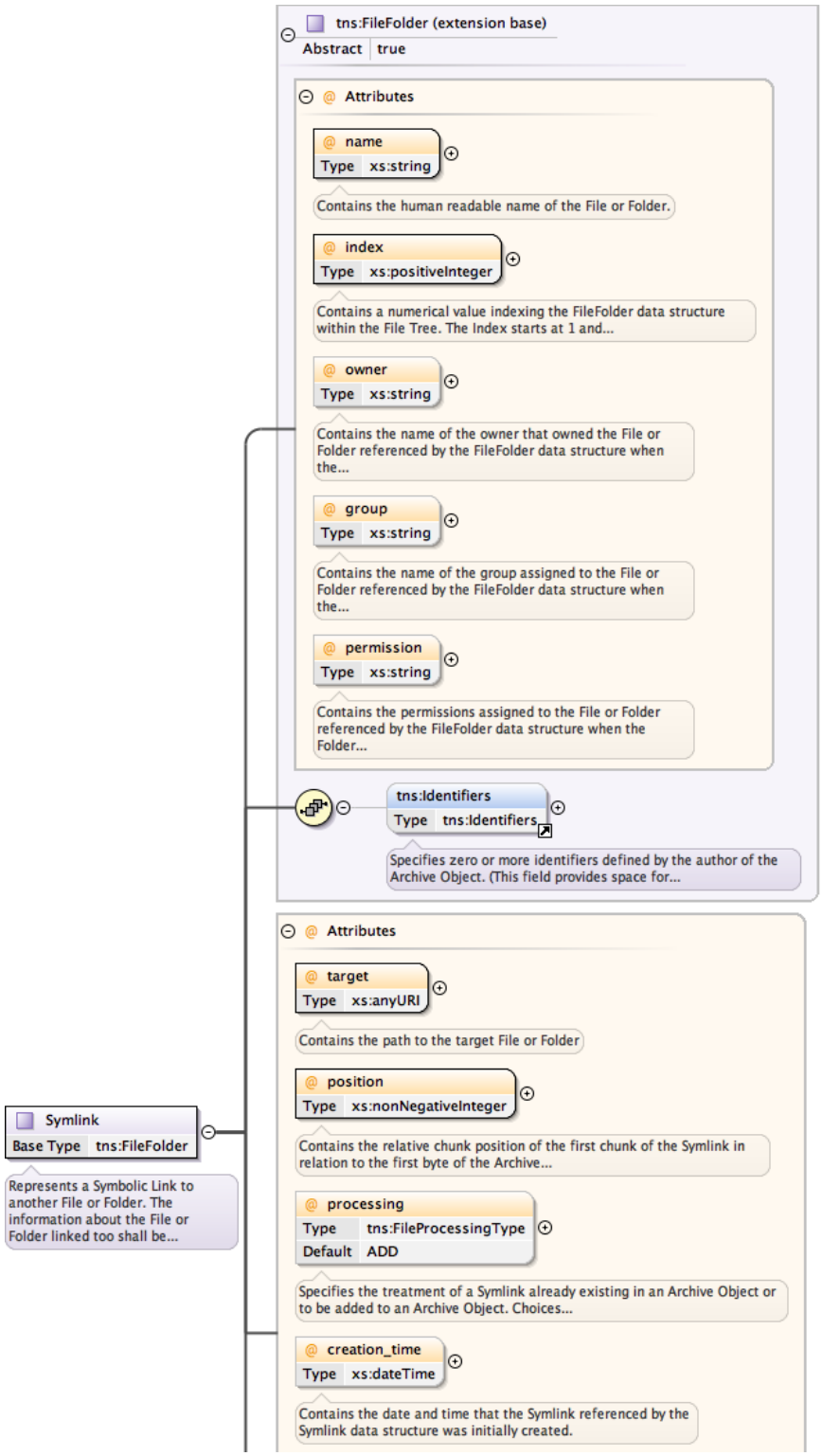
The **File** element is: **Checksums**

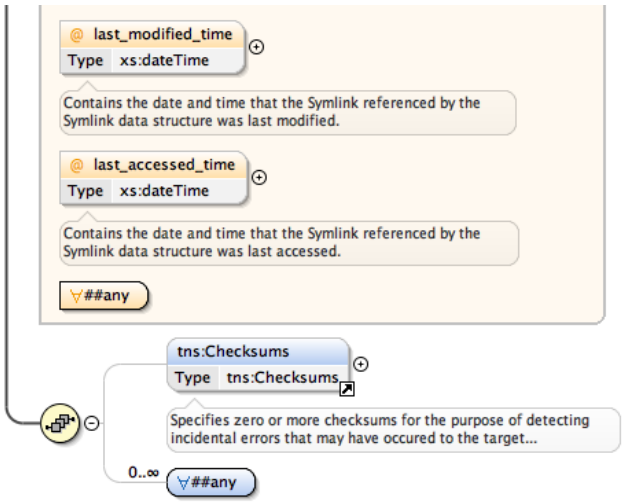
#### 10.12.2.1 Checksums

The **Checksums** element within a **File** data structure shall contain a list of zero or more **Checksum** data types, each of which shall include the value of a checksum of the file referenced by the **File** data structure.

## 10.13 Symlink

The **Symlink** complex data type specifies the target location of a Symbolic Link, for which the name of the Symbolic Link in the FileTree structure is an alias. The target location may be local (internal to the AXF Object) or remote (external to the AXF Object).






```
<xs:complexType name="Symlink">
  <xs:complexContent>
    <xs:extension base="tns:FileFolder">
      <xs:sequence>
        <xs:element ref="tns:Checksums" minOccurs="0" maxOccurs="1" />
        <xs:any minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="target" type="xs:anyURI" use="required" />
      <xs:attribute name="position" type="xs:nonNegativeInteger" use="required" />
      <xs:attribute name="processing" type="tns:FileProcessingType" default="ADD" />
      <xs:attribute name="creation_time" type="xs:dateTime" use="optional" />
      <xs:attribute name="last_modified_time" type="xs:dateTime" use="optional" />
      <xs:attribute name="last_accessed_time" type="xs:dateTime" use="optional" />
      <xs:anyAttribute/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

### 10.13.1 Symlink Attributes

The **Symlink** attributes are: **target, position**, **processing**, **creation_time**, **last_modified_time**, and **last_accessed_time**.

#### 10.13.1.1 target

The **target** attribute shall identify the location of a file or folder for which the name of the Symbolic Link is intended to be an alias. The **target** location shall be either internal to the AXF Object or external. When the **target** location is internal to the AXF Object, the location shall be expressed as another location in the **FileTree** path structure from the location of the Symbolic Link. When the **target** location is external to the AXF Object, the location shall be expressed as a URI.

#### 10.13.1.2 position

The **position** attribute of a **Symlink** data structure shall be a positive integer value identifying the relative Chunk position of the Padding Chunk of the Symbolic Link in relation to the Chunk containing the first byte of the AXF Object.

#### 10.13.1.3 processing

The **processing** attribute of a **Symlink** data structure shall specify the treatment of a Symbolic Link already existing in an AXF Object or to be added to an AXF Object. Choices are ADD, REPLACE, and DELETE. If the attribute is not present, the default treatment shall be ADD. (See Section 9.3.)

#### 10.13.1.4 creation_time

The **creation_time** attribute of a **Symlink** data structure shall contain the date and time at which the **Symlink** data structure initially was created.

#### 10.13.1.5 last_modified_time

The **last_modified_time** attribute of a **Symlink** data structure shall contain the date and time at which the **Symlink** data structure last was modified.

#### 10.13.1.6 last_accessed_time

The **last_accessed_time** attribute of a **Symlink** data structure shall contain the date and time at which the **Symlink** data structure last was accessed.

### 10.13.2 Symlink Elements

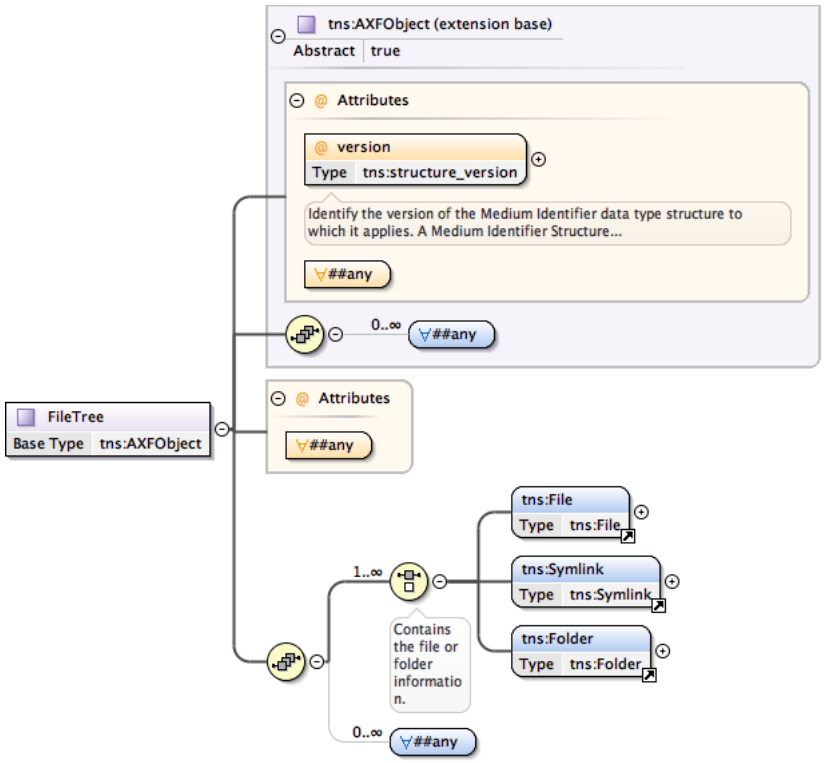
The **Symlink** element is: **Checksums**.

#### 10.13.2.1 Checksums

The **Checksums** element within a **Symlink** data structure shall contain a list of zero or more **Checksum** data types, each of which shall include the value of a checksum of the Padding Chunk preceding the **Symlink** data structure.

## 10.14 FileTree

The **FileTree** complex data type specifies a relative File/Folder hierarchy starting from an arbitrary root.




```
<xs:complexType name="FileTree">
  <xs:complexContent>
    <xs:extension base="tns:AXFObject">
      <xs:sequence>
        <xs:choice minOccurs="1" maxOccurs="unbounded">
          <xs:element ref="tns:File" />
          <xs:element ref="tns:Symlink”/>
          <xs:element ref="tns:Folder" />
        </xs:choice>
        <xs:any minOccurs=”0” maxOccurs=”unbounded”/>
      </xs:sequence>
      <xs:anyAttribute/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

### 10.14.1 FileTree Attributes

The **FileTree** attribute is: **version**.

#### 10.14.1.1 Version

```
<xs:attribute name=“version” type=“tns:structure_version” use=“required” />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the File Tree in which it is found. The File Tree Structure Version applicable to this edition of this standard shall be1.1.

### 10.14.2 FileTree Elements

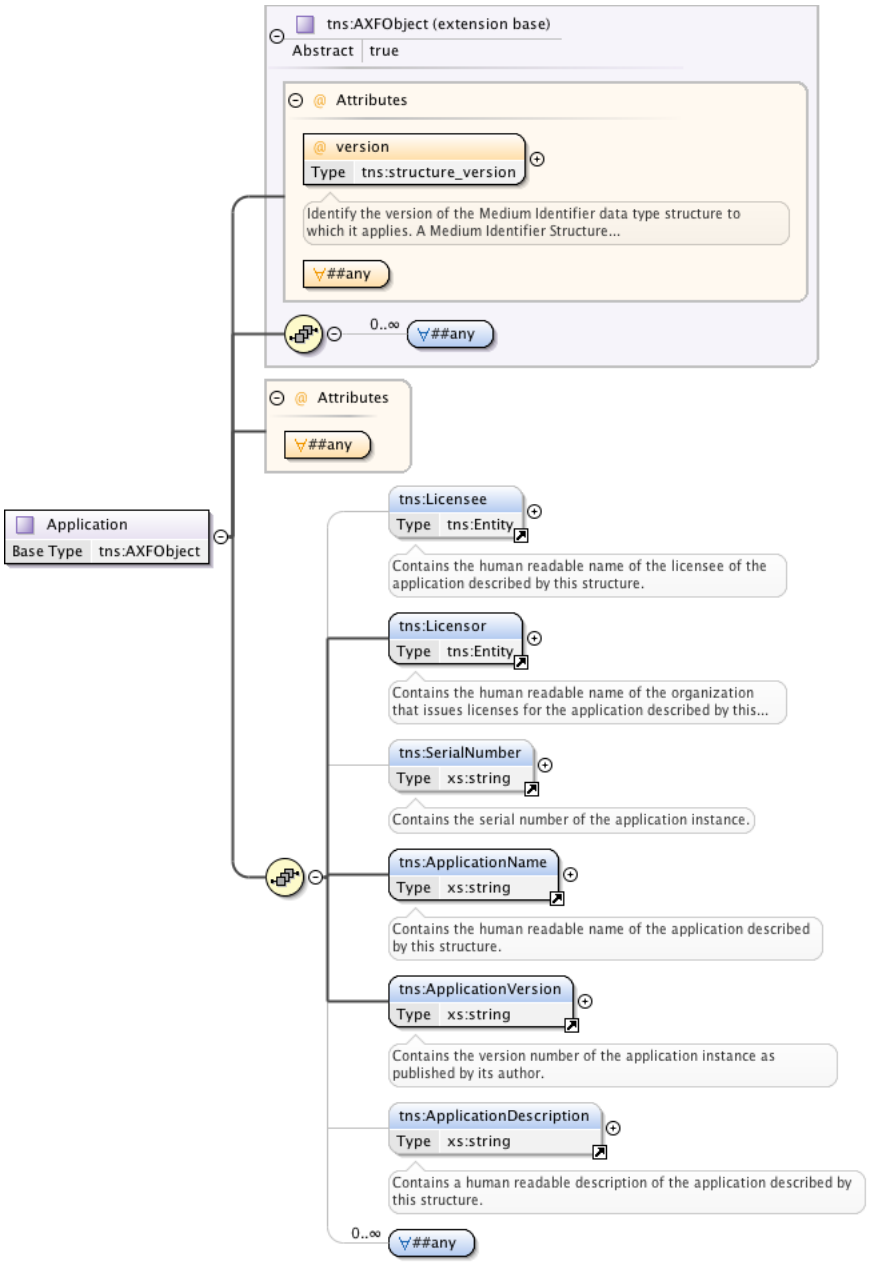
The FileTree elements are: **File, Symlink,** and **Folder**.

#### 10.14.2.1 File, Symlink, or Folder

The **File, Symlink,** and **Folder** elements contained within a **FileTree** data structure shall be **File, Symlink,** or **Folder** data types that provide information about their associated files and folders.

### 10.15 Application

The Application data structure describes a software application that was used in the construction of an AXF Medium or an AXF Object. The fields contained within the Application complex data type shall apply to entities referenced by higher-level data structures into which the Application data type is incorporated.



### 10.15.1 Application Attributes

The Application attribute is: **version**

#### 10.15.1.1 Version

```
<xs:attribute name="version" type="tns:structure_version" use="required" />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the Application data structure in which it is found. The Application Structure Version applicable to this edition of this standard shall be 1.0.

### 10.15.2 Application Elements

The Application elements are: **Licensee**, **Licensor**, **SerialNumber**, **ApplicationName**, **ApplicationVersion**, and **ApplicationDescription**.

#### 10.15.2.1 Licensee

```
<xs:element name="Licensee" type="tns:Entity" />
```

The **Licensee** element contained within an Application data structure shall contain the human readable name of the licensee of the application described by this structure.

Note: The authors of this standard are aware that certain early implementations of the AXF technology included as the **Licensee** element value only a string carrying the name of the licensee instead of the full complement of information implicit in the **Entity** data structure.

#### 10.15.2.2 Licensor

```
<xs:element name="Licensor" type="tns:Entity" />
```

The **Licensor** element contained within an Application data structure shall contain the human readable name of the organization that issues licenses for the application described by this structure.

Note: The authors of this standard are aware that certain early implementations of the AXF technology included as the **Licensor** element value only a string carrying the name of the licensor instead of the full complement of information implicit in the **Entity** data structure.

#### 10.15.2.3 SerialNumber

```
<xs:element name="SerialNumber" type="xs:string" />
```

The **SerialNumber** element contained within an Application data structure shall contain the serial number of the application instance.

#### 10.15.2.4 ApplicationName

```
<xs:element name="ApplicationName" type="xs:string" />
```

The **ApplicationName** element contained within an Application data structure shall contain the human readable name of the application described by this structure.

#### 10.15.2.5 ApplicationVersion

```
<xs:element name="ApplicationVersion" type="xs:string" />
```

The **ApplicationVersion** element contained within an Application data structure shall contain the version number of the application instance as published by its author.

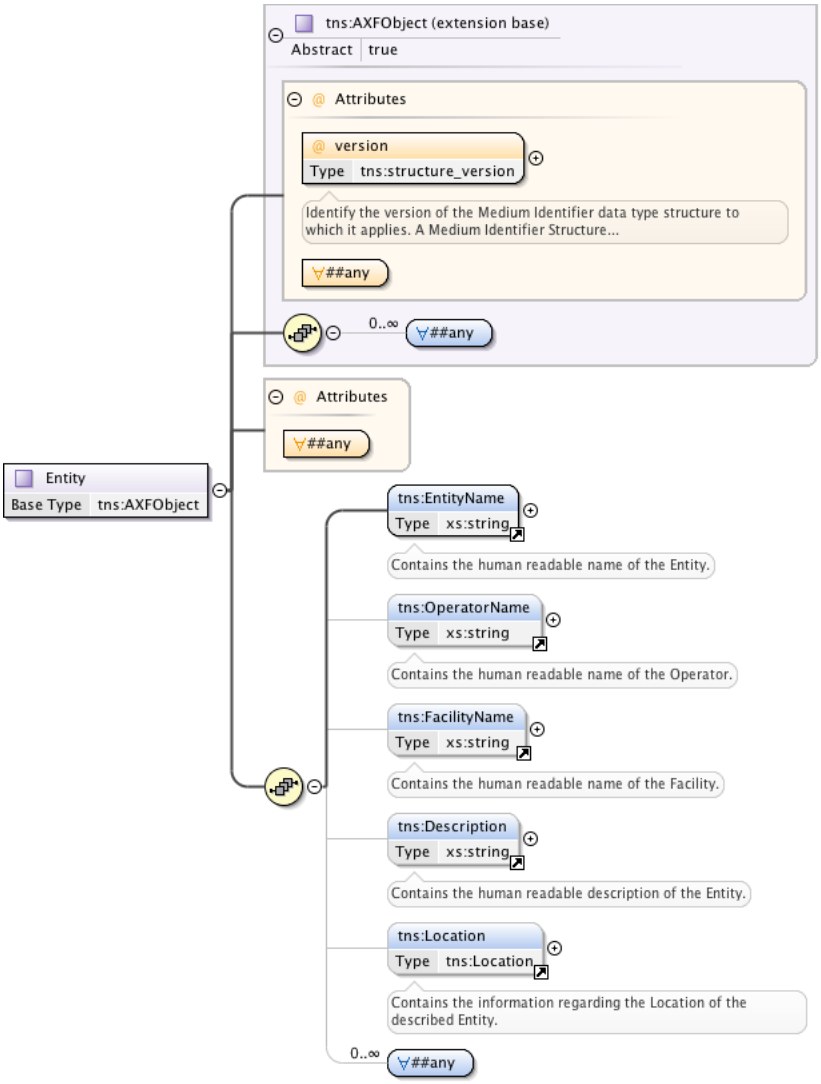
#### 10.15.2.6 ApplicationDescription

```
<xs:element name="ApplicationDescription" type="xs:string" />
```

The **ApplicationDescription** element contained within an Application data structure shall contain a human readable description of the application described by this structure.

### 10.16 Entity

The Entity data structure represents a person or thing. The fields contained within the Entity complex data type shall apply to entities referenced by higher level data structures into which the Entity data type is incorporated.



### 10.16.1 Entity Attributes

The Entity attribute is: **version**.

#### 10.16.1.1 Version

```
<xs:attribute name=“version” type=“tns:structure_version” use=“required” />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the Entity data structure in which it is found. The Entity Structure Version applicable to this edition of this standard shall be 1.0.

### 10.16.2 Entity Elements

The Entity elements are: **EntityName**, **OperatorName**, **FacilityName**, **Description**, and **Location**

#### 10.16.2.1 EntityName

```
<xs:element name="EntityName" type="xs:string" />
```

The **EntityName** element contained within an Entity data structure shall contain the human readable name of the Entity.

#### 10.16.2.2 OperatorName

```
<xs:element name="OperatorName" type="xs:string" />
```

The **OperatorName** element contained within an Entity data structure shall contain the human readable name of the Operator.

#### 10.16.2.3 FacilityName

```
<xs:element name="FacilityName" type="xs:string" />
```

The **FacilityName** element contained within an Entity data structure shall contain the human readable name of the Facility.

#### 10.16.2.4 Description

```
<xs:element name="Description" type="xs:string" />
```

The **Description** element contained within an Entity data structure shall contain the human readable description of the Entity.

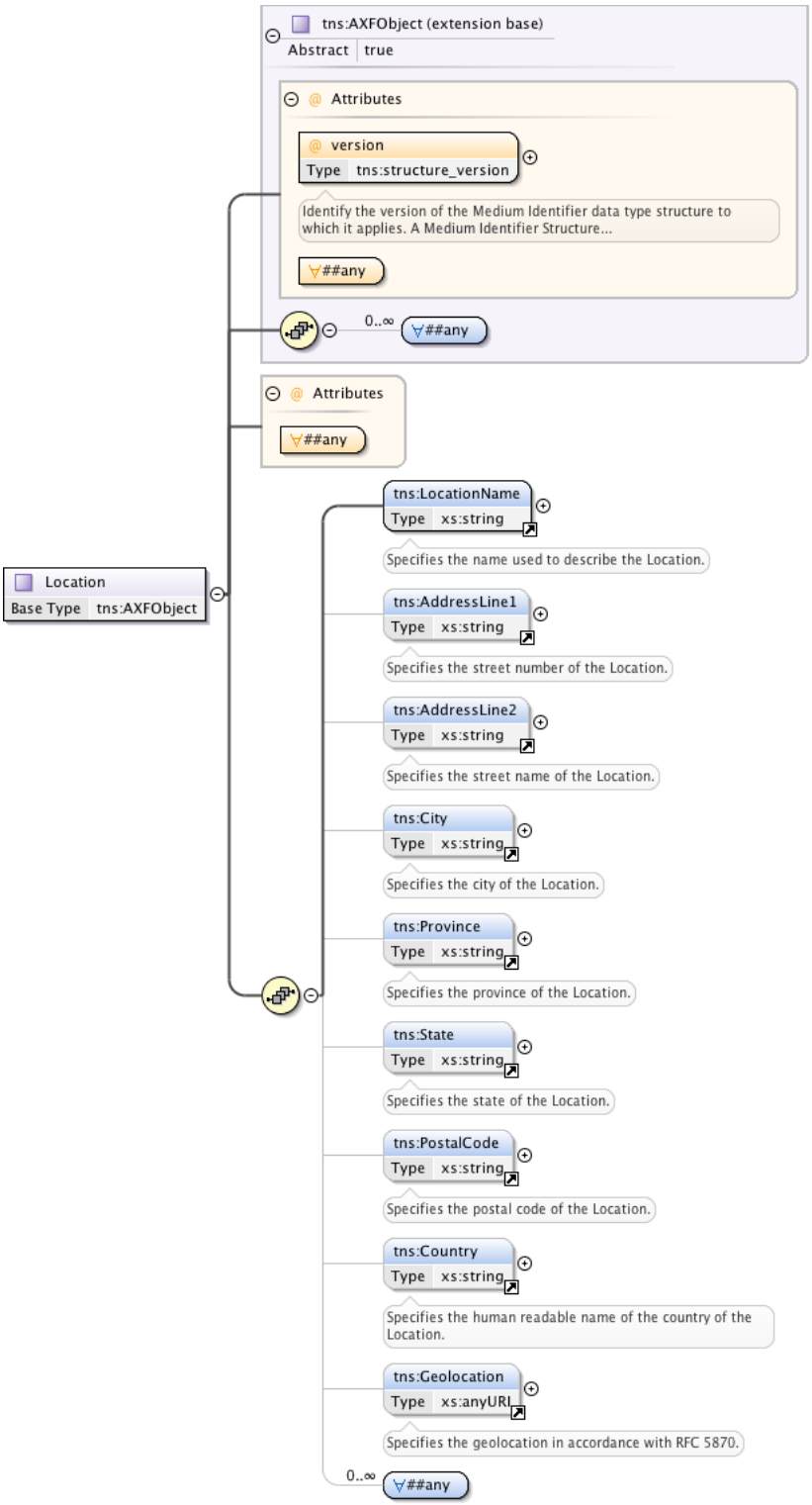
#### 10.16.2.5 Location

```
<xs:element name="Location" type="tns:Location" />
```

The **Location** element contained within an Entity data structure shall contain the information regarding the location of the described Entity.

## 10.17 Location

The fields contained within the Location complex data type shall apply to entities referenced by higher level data structures into which the Location data type is incorporated. Specification of the Geolocation shall be in accordance with RFC 5870 — Uniform Resource Identifier for Geographic Locations.



### 10.17.1 Location Attributes

The Location attribute is: **version**

#### 10.17.1.1 Version

```
<xs:attribute name=“version” type=“tns:structure_version” use=“required” />
```

The **version** attribute shall identify the Structure Version of the data structure used to compose the Location data type in which it is found. The Location Structure Version applicable to this edition of this standard shall be 1.0.

### 10.17.2 Location Elements

The Location elements are: **LocationName, AddressLine1**, **AddressLine2**, **City**, **Province**, **State**, **PostalCode**, **Country**, and **Geolocation**.

#### 10.17.2.1 LocationName

```
<xs:element name="LocationName" type="xs:string" />
```

The **LocationName** element contained within a Location data structure shall specify the name of the Location.

#### 10.17.2.2 AddressLine1

```
<xs:element name="AddressLine1" type="xs:string" />
```

The **AddressLine1** element contained within a Location data structure shall specify the first address line of the Location.

#### 10.17.2.3 AddressLine2

```
<xs:element name="AddressLine2" type="xs:string" />
```

The **AddressLine2** element contained within a Location data structure shall specify the second address line of the Location.

#### 10.17.2.4 City

```
<xs:element name="City" type="xs:string" />
```

The **City** element contained within a Location data structure shall specify the city of the Location.

#### 10.17.2.5 Province

```
<xs:element name="Province" type="xs:string" />
```

The **Province** element contained within a Location data structure shall specify the province of the Location.

#### 10.17.2.6 State

```
<xs:element name="State" type="xs:string" />
```

The **State** element contained within a Location data structure shall specify the state of the Location.

#### 10.17.2.7 PostalCode

```
<xs:element name="PostalCode" type="xs:string" />
```

The **PostalCode** element contained within a Location data structure shall specify the postal code of the Location.

#### 10.17.2.8 Country

```
<xs:element name="Country" type="xs:string" />
```

The **Country** element contained within a Location data structure shall specify the human readable name of the country of the Location.

#### 10.17.2.9 Geolocation

```
<xs:element name="Geolocation" type="xs:anyURI" />
```

The **Geolocation** element contained within a Location data structure shall specify the geolocation in accordance with RFC 5870.

### 10.18 Identifiers

The **Identifiers** data type shall contain zero or more **Identifier** mixed data types referenced by higher level data structures into which the **Identifiers** data type is incorporated.

### 10.19 Checksums

The **Checksums** data type shall contain zero or more **Checksum** mixed data types referenced by higher level data structures into which the **Checksums** data type is incorporated.

#### 10.19.1 Checksum Alias

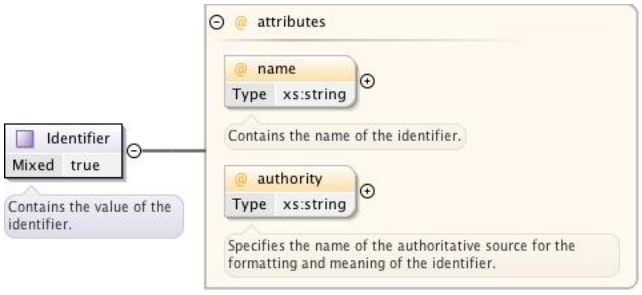
The alias of **Checksums** is **ChecksumTypes**.

#### 10.19.1.1 Checksum Types

The **ChecksumTypes** data type is an alias for the **Checksums** data type that shall contain a list of zero or more **ChecksumType** elements.

### 10.20 Identifier

The fields contained within the **Identifier** mixed data type shall apply to identifiers encapsulated within the **Identifiers** data type. The identifiers contained within **Identifier** elements can be standardized identifiers (e.g., UMID or EIDR) or can be user-defined identifiers (e.g., Tape Number or Database Primary Key).




```
<xs:complexType name="Identifier" mixed="true">
  <xs:attribute name="name" type="xs:string" />
  <xs:attribute name="authority" type="xs:string" />
</xs:complexType>
```

### 10.20.1 Identifier Attributes

The **Identifier** attributes are **Name** and **Authority**.

#### 10.20.1.1 name

The **name** attribute of the **Identifier** data structure shall contain the name of the identifier.

#### 10.20.1.2 authority

The **authority** attribute of the **Identifier** data structure shall specify the name of the authoritative source for the formatting and meaning of the identifier.

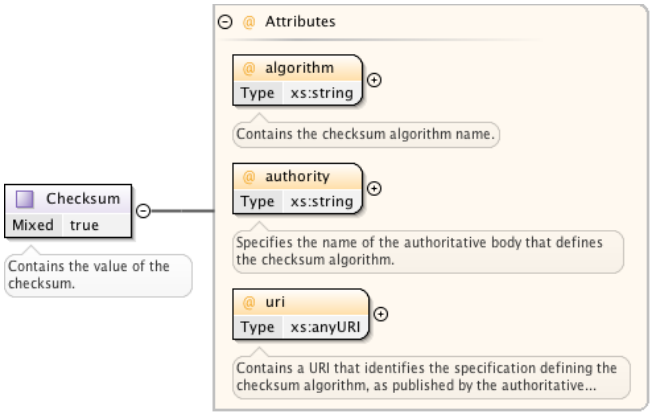
### 10.20.2 Identifier Value

The **value** of the **Identifier** element shall contain the textual value of the identifier.

## 10.21 Checksum

The fields contained within the **Checksum** mixed data type shall carry the product of applying a cryptographic hash algorithm to payload data, plus the identifying name for the algorithm, the authoritative body that published the algorithm, and the document in which the algorithm was published. It shall be encapsulated into the **Checksums** data type for inclusion in higher level structures.

Note: The authors of this standard are aware that some implementations of earlier versions of this standard may generate XML documents that include **Checksum** mixed data types that omit the **authority** and **uri** attributes, that include the value of the computed checksum as an attribute named **value**, and that specify the name of the checksum algorithm as an attribute named **type**.




```
<xs:complexType name="Checksum" mixed="true">
  <xs:attribute name="algorithm" type="xs:string" />
  <xs:attribute name="authority" type="xs:string" />
  <xs:attribute name="uri" type="xs:anyURI" />
</xs:complexType>
```

### 10.21.1 Checksum Attributes

The **Checksum** attributes are: **algorithm, authority, and uri**.

#### 10.21.1.1 algorithm

The **algorithm** attribute of the **Checksum** data structure shall contain the name of the checksum algorithm.

#### 10.21.1.2 authority

The **authority** attribute of the **Checksum** data structure shall contain the name of the authoritative body that published the algorithm identified by the **algorithm** attribute.

#### 10.21.1.3 uri

The **uri** attribute of the **Checksum** data structure shall identify the document published by the authoritative body in which the algorithm identified by the **algorithm** attribute is defined.

Examples: The values shown in the following table can be used to populate the **algorithm**, **authority**, and **uri** attributes of the **Checksum** data type.

| algorithm | authority | uri |
|---|---|---|
| CRC32 | ISO | http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=37010 |
| CRC64 | ISO | http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=37010 |
| MD4 | IETF | http://tools.ietf.org/rfc/rfc1320.txt |
| MD5 | IETF | http://tools.ietf.org/rfc/rfc1321.txt |
| SHA-1 | NIST | http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf |
| SHA-224 | NIST | http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf |
| SHA-256 | NIST | http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf |
| SHA-384 | NIST | http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf |
| SHA-512 | NIST | http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf |
| SHA-512/224 | NIST | http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf |
| SHA-512/256 | NIST | http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf |

### 10.21.2 Checksum Value

The value of the **Checksum** element shall contain the computed value of the checksum determined by application of the algorithm specified by the Checksum **algorithm, authority, and uri** attributes. The value of the Checksum element shall be formatted as a **base64Binary** data type as specified in the W3C XML Schema Part 2 — Datatypes document.

### 10.21.3 Checksum Alias

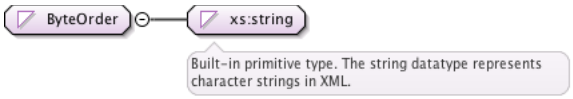
The alias of **Checksum** is **ChecksumType**.

#### 10.21.3.1 Checksum Type

The **ChecksumType** data type is an alias for the **Checksum** data type that shall contain only the **Checksum** attributes.

## 10.22 ByteOrder

The **ByteOrder** simple data type is an enumerated list of the supported byte orderings for Payload Files.




```
<xs:simpleType name="ByteOrder">
  <xs:restriction base="xs:string">
    <xs:enumeration value="LE" />
    <xs:enumeration value="BE" />
    <xs:enumeration value="BOM" />
  </xs:restriction>
</xs:simpleType>
```

### 10.22.1 Byte Order

| Value | Description |
|---|---|
| LE | Little Endian payload |
| BE | Big Endian payload |
| BOM | Byte Order Mark contained within payload |

### 10.23 Media Type

```
<xs:simpleType name="MediaType">
   <xs:restriction base="xs:string">
      <xs:pattern value="([\d\w]{1}[\d\w!#$&amp;\-\^_\.\+]*)\/([\d\w]{1}[\d\w!#$&amp;\-
\^_\.\+]*)(\p{S}.+(=.+)?)*" />
      <xs:maxLength value="255" />
   </xs:restriction>
</xs:simpleType>
```

The Media Type simple data type shall identify the format of the content of the file with which it is associated, using the identification method and value defined in RFC 6838.

### 10.24 Structure Version

```
<xs:simpleType name="structure_version">
   <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]+(.[0-9]+)+"/>
   </xs:restriction>
</xs:simpleType>
```

The Structure Version simple data type shall identify the version of the data structure to which it applies. A Structure Version shall comprise two parts: major and minor version numbers, in the form *[major.minor]*. Structures of the same type having different minor version numbers but the same major version number shall indicate compatible differences between structures; structures having different major version numbers shall indicate incompatible differences between structures.