Linear Tape File System (LTFS) Format Specification

Version 2.5

ABSTRACT: This document defines a Linear Tape File System (LTFS) Format separate from any implementation on data storage media. Using this format, data is stored in LTFS Volumes. An LTFS Volume holds data files and corresponding metadata to completely describe the directory and file structures stored on the volume.

This document has been released and approved by the SNIA. The SNIA believes that the ideas, methodologies and technologies described in this document accurately represent the SNIA goals and are appropriate for widespread distribution. Suggestions for revisions should be directed to http://www.snia.org/feedback/.

SNIA Technical Position

May 19, 2019

USAGE

The SNIA hereby grants permission for individuals to use this document for personal use only, and for corporations and other business entities to use this document for internal use only (including internal copying, distribution, and display) provided that:

1. Any text, diagram, chart, table or definition reproduced shall be reproduced in its entirety with no alteration, and,

2. Any document, printed or electronic, in which material from this document (or any portion hereof) is reproduced, shall acknowledge the SNIA copyright on that material, and shall credit the SNIA for granting permission for its reuse.

Other than as explicitly provided above, you may not make any commercial use of this document or any portion thereof, or distribute this document to third parties. All rights not explicitly granted are expressly reserved to SNIA.

Permission to use this document for purposes other than those enumerated above may be requested by e-mailing tcmd@snia.org. Please include the identity of the requesting individual and/or company and a brief description of the purpose, nature, and scope of the requested use.

All code fragments, scripts, data tables, and sample code in this SNIA document are made available under the following license:

BSD 3-Clause Software License

Copyright (c) 2019, The Storage Networking Industry Association.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of The Storage Networking Industry Association (SNIA) nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

DISCLAIMER

The information contained in this publication is subject to change without notice. The SNIA makes no warranty of any kind with regard to this specification, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The SNIA shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this specification.

Acknowledgements

The SNIA LTFS Technical Working Group, which developed and reviewed this specification, would like to recognize the significant contributions made by the following members:

EMC Corporation..... .....Don Deel
Hewlett Packard Enterprise .....Chris Martin
IBM..... .....David Pease
.....Ed Childers
.....Takeshi Ishimoto
.....Atsushi Abe
NetApp.....David Slik
Oracle Corporation..... .....Matthew Gaffney
.....Carl Madison
Quantum Corporation..... .....Paul Stone
……. Jim Wong
SNIA..... .....Arnold Jones

Contents

List of Figures

List of Tables

1 Introduction

This document defines a Linear Tape File System (LTFS) Format separate from any implementation on data storage media. Using this format, data is stored in LTFS Volumes. An LTFS Volume holds data files and corresponding metadata to completely describe the directory and file structures stored on the volume.

The LTFS Format has these features:

- An LTFS Volume can be mounted and volume content accessed with full use of the data without the need to access other information sources.
- Data can be passed between sites and applications using only the information written to an LTFS Volume.
- Files can be written to, and read from, an LTFS Volume using standard POSIX file operations.

The LTFS Format is particularly suited to these usages:

- Data export and import.
- Data interchange and exchange.
- Direct file and partial file recall from sequential access media.
- Archival storage of files using a simplified, self-contained or “self-describing” format on sequential access media.

2 Scope

This document defines the LTFS Format requirements for interchanged media that claims LTFS compliance. Those requirements are specified as the size and sequence of data blocks and file marks on the media, the content and form of special data constructs (the LTFS Label and LTFS Index), and the content of the partition labels and use of MAM parameters.

The data content (not the physical media) of the LTFS format shall be interchangeable among all data storage systems claiming conformance to this format. Physical media interchange is dependent on compatibility of physical media and the media access devices in use.

NOTE: This document does not contain instructions or tape command sequences to build the LTFS structure.

2.1 Versions

This document describes version 2.5.0 of the Linear Tape File System (LTFS) Format Specification.

The version number for the LTFS Format Specification consists of three integer elements separated by period characters of the form *M.N.R*, where *M*, *N* and *R* are positive integers or zero. Differences in the version number between different revisions of this specification indicate the nature of the changes made between the two revisions. Each of the integers in the format specification are incremented according to Table 1.

Table 1 — Version elements

Element	Description
M	Incremented when a major update has been made to the LTFS Format Specification. Major updates are defined as any change to the on-media format or specification semantics that are expected to break compatibility with older versions of the specification.
N	Incremented when a minor update has been made to the LTFS Format Specification. Minor updates are defined as any change to the on-media format or specification semantics that is not expected to break compatibility with older versions of the specification that have the same value for *M* in the version number.
R	Incremented when textual revisions are made to the LTFS Format Specification. Textual revisions are defined as revisions that improve the clarity of the specification document *without* changing the intent of the document. By definition, minor changes do not alter the on-media format or specification semantics.

NOTE 1: When any element of the specification version number is incremented, all sub-ordinate elements to the right are reset to zero. For example, if the version is 1.0.12 and N is incremented to 1, then R is set to zero resulting in version 1.1.0.

NOTE 2: The first public version of this document used version number 1.0. This value should be interpreted as equivalent to 1.0.0 in the version numbering defined in this document.

The result of comparison between two LTFS version numbers $M_A.N_A.R_A$ and $M_B.N_B.R_B$ is defined in

Table 2.

Table 2 — Version comparisons

Conditional	Description
$M_A < M_B$	$M_A.N_A.R_A$ is an earlier version than $M_B.N_B.R_B$.
$M_A = M_B$ *and* $M_A < N_B$	$M_A.N_A.R_A$ is an earlier version than $M_B.N_B.R_B$.
$M_A = M_B$ *and* $N_A = N_B$ *and* $R_A < R_B$	$M_A.N_A.R_A$ is an earlier version than $M_B.N_B.R_B$. However, as defined above, changes that result only in a different *R* value are descriptive changes in the specification rather than on media changes.

2.2 Conformance

Recorded media claiming conformance to this format shall be in a consistent state when interchanged or stored. See Section 4.1.4.

Any implementation conforming to this specification should be able to correctly read Label and Index structures from all prior versions of this specification and write Label and Index structures conforming to the descriptions in this document. The current Label and Index structures are defined in Section 8 Label Format and in Section 9 Index Format.

NOTE: Where practical, any implementation supporting a given version value for M should endeavor to support LTFS volumes with version numbers containing higher values for N and R than those defined at the time of implementation.

3 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

3.1 Approved references

ISO/IEC 14776-453, SCSI Primary Commands - 3 (SPC-3) [ANSI INCITS.408-2005]

SSC-4 SCSI Stream Commands – 4 [SSC-4] [ANSI INCITS 516-2013]

IETF RFC 4648, The Base16, Base32, and Base64 Data Encodings, http://www.ietf.org/rfc/rfc4648.txt

ISO 8601:2004 Data elements and interchange formats – Information interchange – Representation of dates and times – (UTC)

ISO/IEC 10646:2012: Information technology - Universal Coded Character Set (UCS) (UTF-8)

IETF RFC 4122, Universally Unique Identifier (UUID) URN Namespace http://www.ietf.org/rfc/rfc4122.txt

IETF RFC 3986, Uniform Resource Identifier (URI): Generic Syntax, http://www.ietf.org/rfc/rfc3986.txt

ANSI X3.27-1978 American National Standard Magnetic Tape Labels and File Structure for Information

3.2 References under development

SCSI Primary Commands - 4 (SPC-4) [ANSI INCITS 513:2014]

3.3 Other references

W3C - Extensible Markup Language (XML) http://www.w3.org/XML

NFC – Unicode Normalization Forms - Unicode Standard Annex - UAX#15
http://www.unicode.org/reports/tr15

Unicode Text Segmentation - Unicode Standard Annex - UAX#29 http://www.unicode.org/reports/tr29

OSF CDE 1.1, Remote Procedure Call – Universal Unique Identifier (UUID)
http://pubs.opengroup.org/onlinepubs/9629399/toc.pdf

4 Definitions and Acronyms

For the purposes of this document the following definitions and acronyms shall apply.

4.1 Definitions

4.1.1

Block Position

The position or location of a recorded block as specified by its LTFS Partition ID and logical block number within that partition.

The block position of an Index is the position of the first logical block for the Index.

4.1.2

Complete Partition

An LTFS partition that consists of an LTFS Label Construct and a Content Area, where the last construct in the Content Area is an Index Construct.

4.1.3

Content Area

A contiguous area in a partition, used to record Index Constructs and Data Extents.

4.1.4

Consistent State

A volume is consistent when both partitions are complete and the last Index Construct in the Index Partition has a back pointer to the last Full Index Construct in the Data Partition.

4.1.5

Data Extent

A contiguous sequence of recorded blocks.

4.1.6

Data Partition

An LTFS partition primarily used for data files.

4.1.7

File

A group of logically related extents together with associated file metadata.

4.1.8

Filesystem sync

An operation during which all cached file data and metadata is flushed to the media.

4.1.9

Full Index

A data structure that describes all valid data files in an LTFS volume. The Full Index is an XML document conforming to the XML schema shown in Annex B (normative) LTFS Index XML Schema.

4.1.10

Generation number

A positive decimal integer which shall indicate the specific generation of an Index within an LTFS volume.

4.1.11

Incremental Index

A data structure that describes changes made to the LTFS volume since the last index was written. The Incremental Index is an XML document conforming to the XML schema shown in Annex B (normative) LTFS Index XML Schema.

4.1.12

Index

Either a Full Index or an Incremental Index.

4.1.13

Index Construct

A data construct comprised of an Index and file marks.

4.1.14

Index Partition

An LTFS partition primarily used to store Index Constructs and optionally data files.

4.1.15

Label Construct

A data construct comprised of an ANSI VOL1 tape label, LTFS Label, and tape file marks.

4.1.16

Linear Tape File System (LTFS)

This document describes the Linear Tape File System Format.

4.1.17

LTFS Construct

Any of three defined constructs that are used in an LTFS partition. The LTFS constructs are: Label Construct, Index Construct, and Data Extent.

4.1.18

LTFS Label

A data structure that contains information about the LTFS partition on which the structure is stored. The LTFS Label is an XML document conforming to the XML schema shown in Annex A (normative) LTFS Label XML Schema.

4.1.19

LTFS Partition

A tape partition that is part of an LTFS volume. The partition contains an LTFS Label Construct and a Content Area.

4.1.20

LTFS Volume

A pair of LTFS partitions, one Data Partition and one Index Partition, that contain a logical set of files and directories. The pair of partitions in an LTFS Volume shall have the same UUID. All LTFS partitions in an LTFS volume are *related partitions*.

4.1.21

Medium Auxiliary Memory

An area of non-volatile storage that is part of an individual storage medium. The method of access to this non-volatile storage is standardized as described in the T10/SPC-4 standard.

4.1.22

Partition Identifier (Partition ID)

The logical partition letter to which LTFS data files and Indexes are assigned.

The linkage between LTFS partition letter and physical SCSI partition number is determined by the SCSI partition in which the LTFS Label is recorded. The LTFS partition letter is recorded in the LTFS Label construct, and the SCSI partition number is known by the SCSI positional context where they were read/written.

4.1.23

Sparse file

A file that has some number of empty (unwritten) data regions. These regions are not stored on the storage media and are implicitly filled with bytes containing the value zero (0*x*00).

4.1.24

UUID

Universally unique identifier; an identifier use to bind a set of LTFS partitions into an LTFS volume.

4.1.25

Volume Change Reference (VCR)

A value that represents the state of all partitions on a medium.

4.1.26

Volume Advisory Locking

An indication that the LTFS volume has been locked against future modifications. This is a form of write protection under the control of host software rather than physical hardware.

4.2 Acronyms

ASCII	American Standard Code for Information Interchange
CM	Cartridge Memory
DCE	Distributed Computing Environment
ISO	International Organization for Standardization
LTFS	Linear Tape File System
MAM	Media Auxiliary Memory
NFC	Normalization Form Canonical Composition
OSF	Open Software Foundation
POSIX	Portable Operating System Interface for Unix
T10/SSC-4	ISO/IEC 14776-334, SCSI Stream Commands - 4 (SSC-4) [T10/2123-D]
UTC	Coordinated Universal Time
UTF-8	8-bit UCS/Unicode Transformation Format
UUID	Universally Unique Identifier
W3C	World Wide Web Consortium

XML Extensible Markup Language

5 Volume Layout

An LTFS volume is comprised of a pair of LTFS partitions. LTFS defines two partition types: data partition and index partition. An LTFS volume shall contain exactly one Data Partition and exactly one Index Partition.

5.1 LTFS Partitions

Each partition in an LTFS volume shall consist of a Label Construct followed by a Content Area. This logical structure is shown in Figure 1.

Figure 1 — LTFS Partition

The Label Construct is described in Section 5.2 LTFS Constructs and in Section 8 Label Format. The Content Area contains some number of interleaved Index Constructs and Data Extents. These constructs are described in Section 5.2 LTFS Constructs and in Section 9 Index Format. The precise layout of the partitions is defined in Section 5.3 Partition Layout.

5.2 LTFS Constructs

LTFS constructs are comprised of file marks and records. These are also known as 'logical objects' as found in T10 SSC specifications and are not described here. An LTFS volume contains three kinds of constructs.

- A Label Construct contains identifying information for the LTFS volume.
- A Data Extent contains file data written as sequential logical blocks. A file consists of zero or more Data Extents plus associated metadata stored in the Index Construct.
- An Index Construct contains an Index, which is an XML data structure which describes the mapping between files and Data Extents.

5.2.1 Label Construct

Each partition in an LTFS volume shall contain a Label Construct with the following structure. As shown in Figure 2, the construct shall consist of an ANSI VOL1 label, followed by a single file mark, followed by one record in LTFS Label format, followed by a single file mark. Each Label construct for an LTFS volume shall contain identical information except for the "location" field of the LTFS Label.

The content of the ANSI VOL1 label and the LTFS Label is specified in Section 8 Label Format.

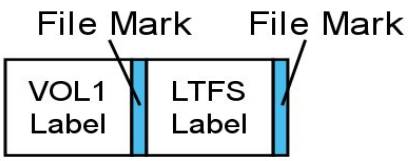


Figure 2 — Label Construct

5.2.2 Data Extent

A Data Extent is a set of one or more sequential logical blocks used to store file data. The “blocksize” field of the LTFS Label defines the block size used in Data Extents. All blocks within a Data Extent shall have this fixed block size except the last block, which may be smaller.

The use of Data Extents to store file data is specified in Section 6 Data Extents.

5.2.3 Index Construct

Figure 3 shows the structure of an Index Construct. An Index Construct consists of a file mark, followed by an Index, followed by a file mark. An Index consists of a record that follows the same rules as a Data Extent, but it does not contain file data. That is, the Index is written as a sequence of one or more logical blocks of size “blocksize” using the value stored in the LTFS Label. Each block in this sequence shall have this fixed block size except the last block, which may be smaller. This sequence of blocks records the Index XML data that holds the file metadata and the mapping from files to Data Extents. The Index XML data recorded in an Index Construct shall be written from the start of each logical block used. That is, Index XML data may not be recorded offset from the start of the logical block.

Figure 3 — Index Construct

Indexes also include references to other Indexes in the volume. References to other Indexes are used to maintain consistency between partitions in a volume. These references (back pointers and self pointers) are described in Section 5.4 Index Layout.

The content of the Index is described in Section 9 Index Format.

5.3 Partition Layout

This section describes the layout of an LTFS Partition in detail. An LTFS Partition contains a Label Construct followed by a Content Area. The Content Area contains zero or more Data Extents and Index Constructs in any order. The last construct in the Content Area of a complete partition shall be an Index Construct.

Figure 4 illustrates an empty complete partition. It contains a Label Construct followed by an Index Construct. This is the simplest possible complete partition.

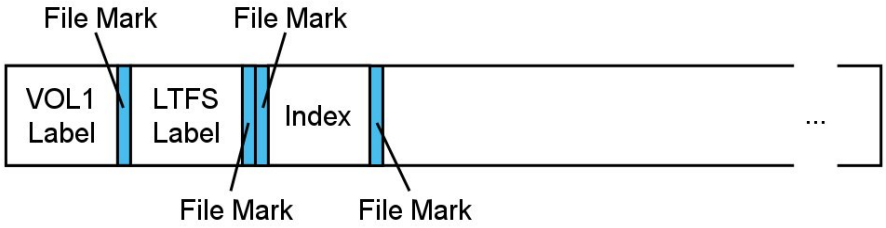


Figure 4 — Partition Layout

Figure 5 illustrates a complete partition containing data. The Content Area on the illustrated partition contains two Data Extents (the first extent comprising the block 'A', the second extent comprising blocks 'B' and 'C') and three Index Constructs.

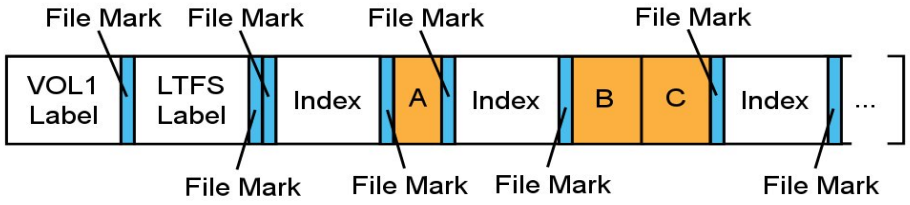


Figure 5 — Complete partition containing data

NOTE: There must not be any additional data trailing the end of the VOL1 Label, the LTFS Label, nor any Index on an LTFS Volume. The Label Construct must be recorded starting at the first logical block in each partition.

5.4 Index Layout

Each Index data structure contains information used to verify the consistency of an LTFS volume.

- A generation number, which records the age of this Index relative to other Indexes in the volume.
- A self pointer, which records the volume to which the Index belongs and the block position of the Index within that volume.
- A back pointer, which records the block position of the last Full Index present on the Data Partition immediately before this Index was written.
- An optional second back pointer, which will only be present if an Incremental Index has been written since the last Full Index on the Data Partition. If present this pointer records the block position of the most recent Incremental Index.

5.4.1 Generation Number

Each Index in a volume has a generation number, a non-negative integer that increases as changes are made to the volume. In any consistent LTFS volume, the Index with the highest generation number on the volume represents the current state of the entire volume and must be a Full Index. Generation numbers are assigned in the following way:

- Given two Indexes on a partition, the one with a higher block position shall have a generation number greater than or equal to that of the one with a lower block position.
- Two Indexes in an LTFS volume may have the same generation number if and only if their contents are identical except for these elements:
  - access time values for files and directories (described in Section 9.2 Index),
  - the self pointer (described in Section 5.4.2 Self Pointer), and
  - the back pointer (described in Section 5.4.3 Back Pointer).

NOTE: The value of the generation number between any two successive Indexes may increase by any positive integer value. That is, the magnitude of increase between any two successive Indexes is not assumed to be equal to 1.

The first Index on an LTFS Volume shall be generation number '1'.

5.4.2 Self Pointer

The self pointer for an Index is comprised of the following information:

- The UUID of the volume to which the Index belongs
- The block position of the Index

The self pointer is used to distinguish between Indexes and Data Extents. An otherwise valid Index with an invalid self pointer shall be considered a Data Extent for the purpose of verifying that a volume is valid and consistent. This minimizes the likelihood of accidental confusion between a valid Index and a Data Extent containing Index-like data.

5.4.3 Back Pointer

Each Index contains at most two back pointers, defined as follows.

- If the Index resides in the Data Partition, the full index back pointer shall contain the block position of the preceding Full Index in the Data Partition. If no preceding Index exists, no back pointer shall be stored in this Index. Back pointers are stored in the Index as described in Section 9.2 Index.
- If the Index resides in the Index Partition and has generation number N then the full index back pointer for the Index shall contain either the block position of a Full Index having generation number N in the Data Partition, or the block position of the last Full Index having at most generation number N−1 in the Data Partition. If no Index of generation number N-1 or less exists in the Data Partition, then the Index in the Index Partition is not required to store a back pointer.
- On a consistent volume, the final Index in the Index Partition shall contain a back pointer to the final Full Index in the Data Partition.
- On a volume containing Incremental Indexes, an index residing in the Data Partition may contain a second back pointer with the block position of the most recent Incremental Index on the Data Partition written since the last Full Index
- As a consequence of the rules above, no Index may contain a back pointer to itself or to an Index with a higher generation number.

On a consistent volume, the rules above require that the Indexes on the Data Partition and the final Index on the Index Partition shall form an unbroken chain of back pointers. Figure 6 illustrates this state for a volume not containing any Incremental Indexes, and Figure 7 illustrates the corresponding state for a volume which does contain Incremental Indexes. See Section 9 for more detail on full and incremental indexes.

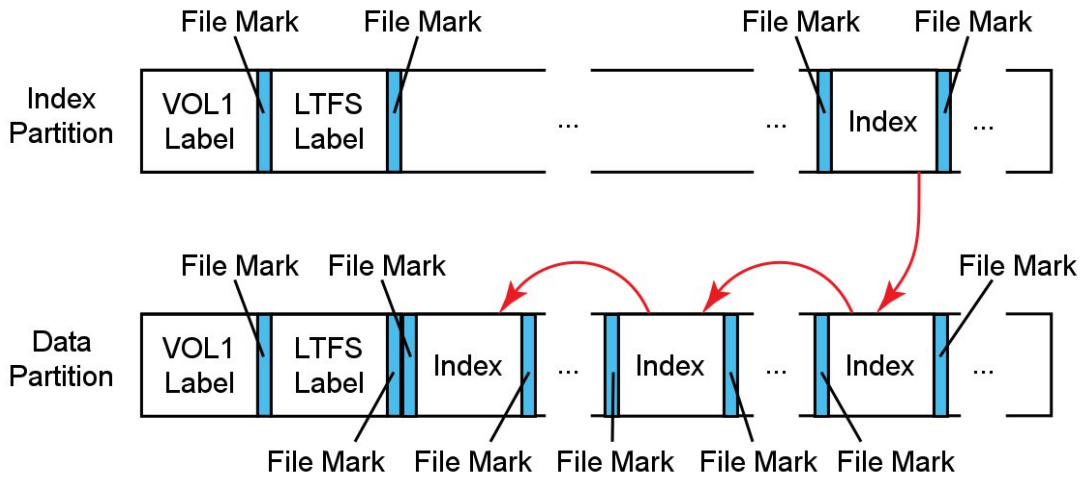


Figure 6 — Back Pointer example

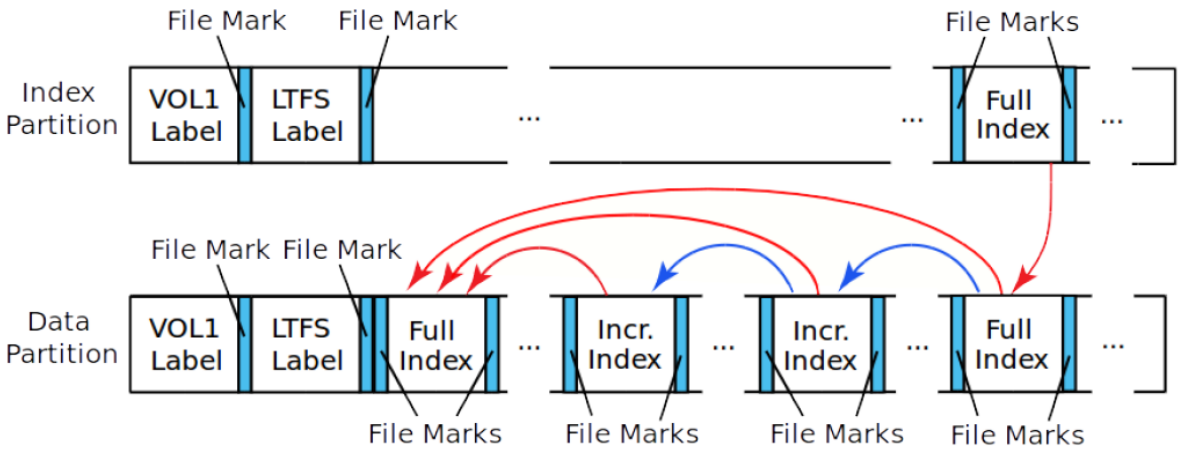


Figure 7 — Back Pointer example for Incremental Indexes

In **Figure 7** the red arrows represent back pointers to a Full Index, and the blue arrows represent back pointers to an Incremental Index.

6 Data Extents

A Data Extent is a set of one or more sequential records subject to the conditions listed in Section 5.2.2 Data Extent. This section describes how files are arranged into Data Extents for storage on an LTFS volume. Logically, a file contains a sequence of bytes; the mapping from file byte offsets to block positions is maintained in an Index. This mapping is called the extent list.

6.1 Extent Lists

A file with zero size has no extent list.

Each entry in the extent list for a file encodes a range of bytes in the file as a range of contiguous bytes in a Data Extent. An entry in the extent list is known as an extent. Each entry shall contain the following information:

- **partition ID** – partition that contains the Data Extent comprising this extent.
- **start block** (start block number) – block number within the Data Extent where the content for this extent begins.
- **byte offset** (offset to first valid byte) – number of bytes from the beginning of the start block to the beginning of file data for this extent. This value shall be strictly less than the size of the start block. The use of byte offset is described in Section 6.2.3 Starting and ending Data Extent in mid-block.
- **byte count** – number of bytes of file content in this Data Extent.
- **file offset** – number of bytes from the beginning of the file to the beginning of the file data recorded in this extent.

NOTE: Version 1.0 of this specification did not explicitly include file offsets in the extent list. When interpreting LTFS Volumes written based on the Version 1.0 specification, the file offsets shall be determined as follows.

- The first extent list entry begins at file offset 0.
- If an extent list entry begins at file offset N and contains K bytes, the following extent list entry begins at file offset N + K.

These file extent rules for version 1.0 of the specification necessarily imply that the order of extents recorded in the Index shall be preserved during any subsequent update of the Index to another version 1.0 Index.

The inclusion of the File Offset value for each extent starting from version 2.0.0 of this specification removes the significance of the order in which extents are recorded in the Index.

Implementers are encouraged to record extents in the same logical order as they exist in the represented file.

In the extent list for any file, no extent may contain bytes that extend beyond the logical end of file. The logical end of file is defined by the file length recorded in the Index. Also, in any extent list for any file, there shall not exist any pair of extents that contain overlapping logical file offsets. That is, no extent is allowed to logically overwrite any data stored in another extent.

An extent list entry shall be a byte range within a single Data Extent; that is, it shall not cross a boundary between two Data Extents. This requirement allows a deterministic mapping from any file offset to the block position where the data can be found. On the other hand, two extent list entries (in the same file or in different files) may refer to the same Data Extent.

6.2 Extents Illustrated

This section illustrates various forms of extent list entries and the mapping from files to these extents. The illustrations are not exhaustive. Other combinations of starting and ending blocks are possible.

The LTFS Partition ID is an essential element of an extent definition. For simplicity, the LTFS Partition ID and File Offset are not shown explicitly in the extents lists illustrated in Table 3 — Extent list entry starting and ending with full block, Table 4, and Table 5 — Extent list entry starting and ending in mid-block. Note that not all extents in an extent list shall be on the same partition.

6.2.1 Starting and ending Data Extent with full block

Figure 8 illustrates an extent of 3 full size blocks contained within a Data Extent of 3 blocks, *N* through *N* + 2.

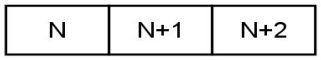


Figure 8 — Extent starting and ending with full block

The extent list entry for this extent is shown in Table 3.

Table 3 — Extent list entry starting and ending with full block

Start Block	Offset	Length
N	0	3 × *Blk*

NOTE: Blk is the length of a full-sized block.

6.2.2 Starting Data Extent with full block and ending with fractional block

Figure 9 illustrates an extent of 2 full-size blocks and one fractional block of *K* bytes, contained within a Data Extent of 2 full size blocks *N* and *N* + 1 and one fractional block *N* + 2.

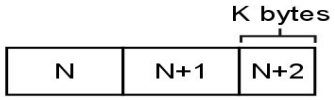


Figure 9 — Extent starting with full block and ending with fractional block

The extent list entry for this extent is shown in Table 4.

Table 4 — Extent list entry starting with full block and ending with fractional block

Start Block	Offset	Length
N	0	(2 × *Blk*) + *K*

NOTE: **K** is the length of the fractional block, where **K < Blk**

6.2.3 Starting and ending Data Extent in mid-block

Figure 10 illustrates an extent smaller than 3 blocks, contained within a Data Extent of 3 full size blocks. Valid data begins in block *N* at byte number *J* and continues to byte number *K* of block *N* + 2. The last block of the extent, block *N* + 2, may be a fractional block.

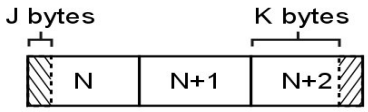


Figure 10 — Extent starting and ending in mid-block

The extent list entry for this extent is shown in Table 5.

Table 5 — Extent list entry starting and ending in mid-block

Start Block	Byte Offset	Byte Count
N	*J*	(*Blk* − *J*) + *Blk* + *K*

6.3 Files Illustrated

This section illustrates various possible extent lists for files. These illustrations are not exhaustive; other combinations of extent geometry and ordering are possible. The extents shown in this section are always displayed in file offset order, but they may appear in any order on a partition, or even in different partitions. As in the previous section, Partition IDs are omitted for simplicity. Unless otherwise noted these examples illustrate non-sparse files that have all file data written to the media.

6.3.1 Simple Files

Figure 11 illustrates a file contained in a single Data Extent of three blocks. The data fills the first two blocks and *K* bytes in the last block. The last block of the extent, block *N* + 2, may be a fractional block. This file is recorded as a regular (non-sparse) file. See Table 6.

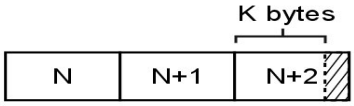


Figure 11 — File contained in a single Data Extent

Table 6 — Extent list entry for file contained in a single Data Extent

Start Block	Byte Offset	Byte Count	File Offset
N	0	(2 × *Blk*) + *K*	0

Figure 12 illustrates a file contained in two Data Extents of three blocks each. The data fills the first two blocks of extent *N* and *K* bytes of block *N* + 2, and the first two blocks of extent *M* and *L* bytes of block *M* + 2. The last block of each extent, block *N* + 2 and *M* + 2, may be fractional blocks. This file is recorded as a regular (non-sparse) file. Table 7 shows file details.

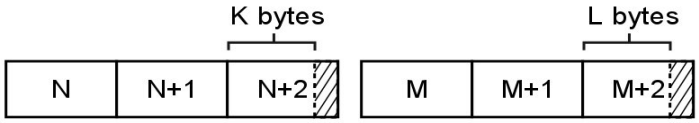


Figure 12 — File contained in two Data Extents

Table 7 — Extent list entry for a file contained in two Data Extents

Start Block	Byte Offset	Byte Count	File Offset
N	0	(2 × *Blk*) + *K*	0
M	0	(2 × *Blk*) + *L*	(2 × *Blk*) + *K*

6.3.2 Shared Blocks

Figure 13 illustrates two full-sized blocks which are referenced by three files. Blocks may be shared among multiple files to improve storage efficiency. File 1 uses the first *K* bytes of block *N* . File 2 uses *Q* bytes in the mid part of block *N* , and (*Blk* − *R*) bytes at the end of block *N* + 1. File 3 uses the last (*Blk* − *P* − *Q*) bytes at the end of block *N* and the first *T* bytes of block *N* + 1.

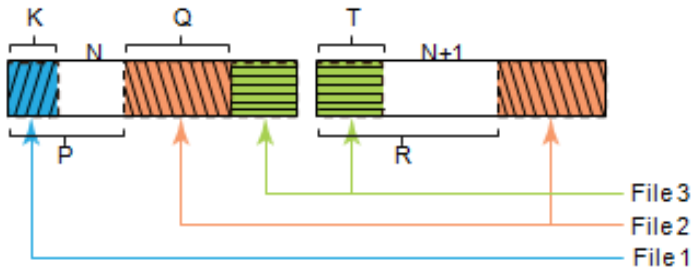


Figure 13 — Shared Blocks example

The extent lists for files 1, 2, and 3 are shown in Table 8.

Table 8 — Extent lists for Shared Blocks example

	Start Block	Byte Offset	Byte Count	File Offset
File 1	*N*	0	*K*	0
File 2	*N*	*P*	*Q*	0
	N+1	*R*	*Blk − R*	*Q*
File 3	*N*	*P + Q*	*Blk − P − Q + T*	0

NOTE: If N were a fractional block, File 3 would map to two entries in the extent list. As illustrated, block **N** is a full block, and File 3 may be mapped to the single extent list entry shown above. Alternatively, because blocks may always be treated as independent Data Extents, File 3 could be mapped to two entries in the extent list, one entry per block (**N** and **N + *1***).

6.3.3 Sparse Files

The length of a file, as recorded in the Index, may be greater than the total size of data encoded in that file's extent list. A file may also have non-zero size but no extent list. In both of these cases, all bytes not encoded in the extent list shall be treated as zero (0*x*00) bytes.

Figure 14 illustrates a sparse file that is contained in two Data Extents. In this figure, all white areas of the file are filled with bytes that are set to zero (0*x*00). The file starts with *T* bytes with value zero(0*x*00). The first extent stores *K* bytes of data which fills the file from byte *T* to *T* + *K*. The file contains *R* bytes with value zero (0*x*00) from file offset *T* + *K* to *T* + *K* + *R*. The second extent contains *Q* file bytes representing the file content from file offset *T* + *K* + *R* to *T* + *K* + *R* + *Q*. The end of the file from file offset *T* + *K* + *R* + *Q* is filled with bytes set to value zero (0*x*00) to the defined file size *P.*

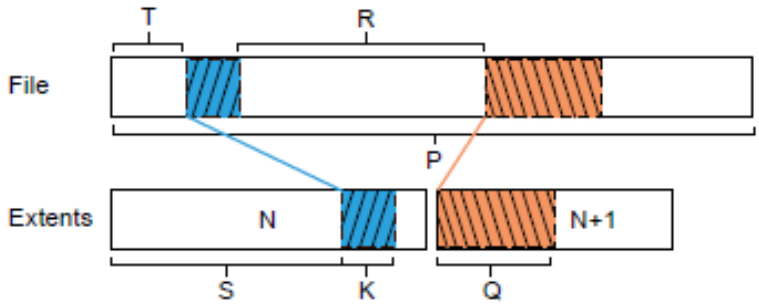


Figure 14 — Sparse files example

The extent list for this file is shown in Table 9.

Table 9 — Extent list for sparse files example

Start Block	Byte Offset	Byte Count	File Offset
N	S	*K*	*T*
N + 1	0	*Q*	*T* + *K* + *R*

NOTE 1: Version 1.0 of this specification, implied zeros could only appear at the end of a file; other types of sparse files were not supported. When appending to the end of a file that is to be stored on a volume in compliance with version 1.0 of this specification, any implied trailing zero bytes in the file must be explicitly written to the media to avoid leaving holes in the extent list for the file.

NOTE 2: Version 1.0 of this specification did not support sparse files.

6.3.4 Shared Data

Figure 15 illustrates four Data Extents which are partly shared by two files. Overlapping extent lists may be used to improve storage efficiency.

NOTE: Methods to implement data deduplication are beyond the scope of this document. Implementations must read files with overlapping extent lists correctly, but they are not required to generate such extent lists.

In Figure 15, File 1 uses all blocks in Data Extents *N* , *M* , and *R*. File 2 uses some of the blocks in Data Extents *N* , *R* and *V* . The extent lists for the two files are shown in Table 10. The two files share some of the data in blocks *N* , *N* + 1, *N* + 2, *R* + 1 and *R* + 2.

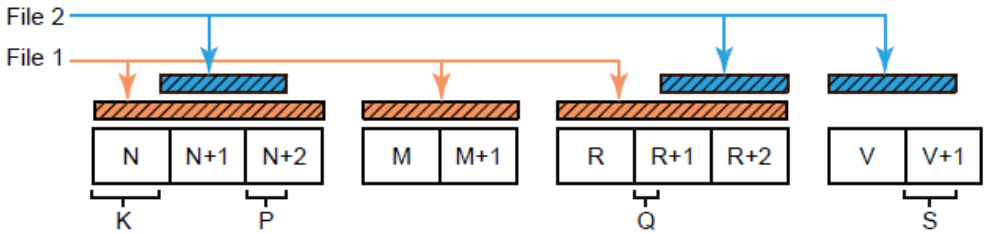


Figure 15 — Shared data example

The extent lists for files 1 and 2 are shown in Table 10.

Table 10 — Extent lists for shared data example

	Start Block	Byte Offset	Byte Count	File Offset
File 1	*N*	0	3 × *Blk*	0
	M	0	2 × *Blk*	3 × *Blk*
	R	0	3 × *Blk*	(3 × *Blk*) + (2 × *Blk*)
File 2	*N*	*K*	(*Blk* − *K*) + *Blk* + *P*	0
	R+1	*Q*	(*Blk* − *Q*) + *Blk* *Blk* +	(*Blk* − *K*) + *Blk* + *P*
	V	0	*S*	(*Blk* − *K*) + *Blk* + *P* + (*Blk* − *Q*) + *Blk*

7 Data Formats

The LTFS Format uses the data formats defined in this section to store XML field values in the Index Construct and Label Construct.

7.1 Boolean format

Boolean values in LTFS structures shall be recorded using the values: “true”, “1”, “false”, and “0”. When set to the values “true” or “1”, the boolean value is considered to be set and considered to evaluate to true. When set to the values “false” or “0”, the boolean value is considered to be unset, and considered to evaluate to false.

7.2 Creator format

LTFS creator values shall be recorded in conformance with the string format defined in Section 7.6 String format with the additional constraints defined in this section.

LTFS creator values shall be recorded as a Unicode string containing a maximum of 1024 Unicode code points. The creator value shall include product identification information, the operating platform, and the name of the executable that wrote the LTFS volume.

An example of the recommended content for creator values is:

IBM LTFS 1.2.0 - Linux - mkltfs

The recommended format for a creator value is a sequence of values separated by a three character separator. The separator consists of a space character, followed by a hyphen character, followed by another space character. The recommended content for the creator value is *Company Product Version* - *Platform* - *binary name* where definitions are as defined in Table 11.

Table 11 — Creator format definitions

Symbol	Description
Company Product Version	Identifies the product that created the volume.
Platform	Identifies the operating system platform for the product.
binary name	Identifies the executable that created the volume.

Any subsequent data in the creator format should be separated from this content by a hyphen character.

7.3 Extended attribute value format

An extended attribute value shall be recorded as one of two possible types:

1. The “text” type shall be used when the value of the extended attribute conforms to the format described in Section 7.6 String format. The encoded string shall be stored as the value of the extended attribute and the type of the extended attribute shall be recorded as “text”.
2. The “base64” type shall be used for all values that cannot be represented using the “text” type. Extended attribute values stored using the “base64” type shall be encoded as base64 according to RFC 4648, and the resulting string shall be recorded as the extended attribute value with the type recorded as “base64”. The encoded string may contain whitespace characters as defined by the W3C Extensible Markup Language (XML) 1.0 standard (space, tab, carriage return, and line feed). These characters shall be ignored when decoding the string.

7.4 Name format

File and directory names, and extended attribute keys in an LTFS Volume shall conform to the naming rules in this section.

Names shall be valid Unicode and shall be 255 code points or less after conversion to Normalization Form C (NFC). Names shall be stored in a case-preserving manner. Since names are stored in an Index, they shall be encoded as UTF-8 in NFC. Names may include any characters allowed by the W3C Extensible Markup Language (XML) 1.0 standard except for the those listed in Table 12.

Table 12 — Reserved characters for name format

Character	Description
U+002F	slash
U+003A	colon

Note that the null character U+0000 is disallowed by *W3C XML 1.0*. See *W3C XML 1.0* for a full list of disallowed characters. The characters listed in Table 13 are allowed, but they should be avoided for reasons of cross-platform compatibility.

Table 13 — Characters which should be avoided for name format

Character	Description
U+0009, U+000A and U+000D	control codes
U+0022	double quotation
U+002A	Asterisk
U+003F	question mark
U+003C	less than sign
U+003E	greater than sign
U+005C	Backslash
U+007C	vertical line

Implementations which claim compliance with version 2.4.0 or later of this specification shall support the percent-encoding of names as described below in order to avoid issues with the characters listed in Table 12 above.

Percent-encoding is described in IETF RFC3986. Reserved characters are replaced by a triplet consisting of the percent character '%' followed by the two hexadecimal digits representing the character's numeric value. For example the colon character (':', U+003A) would be represented as the string "%3A". In accordance with RFC3986, this further means that for any names that already contain the percent character, and for which percent-encoding is enabled, that percent character itself needs to be encoded as the triplet "%25" (since the percent character is encoded as 0x25). Also in accordance with RFC3986 uppercase hexadecimal digits should be used for all percent-encodings, although lowercase digits 'a' through 'f' shall be treated as equivalent to their uppercase equivalents 'A' through 'F'.

Table 14 shows some examples of the encoding:

Table 14 — Name percent-encoding

Source name	Encoded name	Description
Testfile1.txt	Testfile1.txt	No transformation necessary
Testfile:1.txt	Testfile%3A1.txt	Colon must be encoded since it is reserved
Testfile%3A.txt	Testfile%253A.txt	Percent must be encoded to avoid ambiguity
Testfile:%1.txt	Testfile%3A%251.txt	Both colon and percent characters encoded
com.my.co:some_xattr	com.my.co%3Asome_xattr	Extended attribute name must be encoded

Names which have been processed using percent-encoding are indicated by the inclusion of the attribute tag **percentencoded**. If a name element includes this attribute tag, the value of the tag shall contain a value conforming to the boolean format definition provided in Section 7.1 Boolean format.

When the **percentencoded** attribute tag is present and has the value true, the corresponding name has been processed to replace one or more characters with an encoded triplet as described above. When reading back from the volume, the inverse operation should be applied to transform the name back into its original form. In cases where the underlying operating system does not support the characters in their original form, an implementation may choose to use the transformed name or to report an error to the user.

If the **percentencoded** attribute tag does not exist, or has the value false, then the name encoding transformation shall not be performed when writing to or reading from the volume.

Note that if the name does not contain any encoded triplets then it is strongly recommended that the **percentencoded** attribute tag should be omitted rather than including it with the value false.

See Sections 9.2.7, 9.2.9 and 9.2.10 for further details. Annex G contains informative tables showing how various characters are represented in an index.

7.5 Name pattern format

File name patterns in data placement policies shall be valid names as defined in Section 7.4 Name format. A file name pattern shall be compared to a file name using these rules:

1. Comparison shall be performed using canonical caseless matching as defined by the Unicode Standard, except for the code points U+002A and U+003F.
2. Matching of name patterns to file names shall be case insensitive.
3. U+002A (asterisk ‘*’) shall match zero or more Unicode grapheme clusters.
4. U+003F (question mark ‘ ?’) shall match exactly one grapheme cluster.

For more information on grapheme clusters, see *Unicode Standard Annex 29, Unicode Text Segmentation*.

7.6 String format

A character string encoded using UTF-8 in NFC. The string shall only contain characters allowed in element values by the W3C Extensible Markup Language (XML) 1.0 specification.

7.7 Time stamp format

Time stamps in LTFS data structures shall be specified as a string conforming to the ISO 8601 date and time representation standard. The time stamp shall be specified in UTC (Zulu) time as indicated by the ‘Z’ character in this example:

2013-02-01T18:35:47.866846222Z

The time shall be specified with a fractional second value that defines 9 decimal places after the period in the format.

The general time format is *YYYY-MM-DD*T*hh*:*mm*:*ss*.*nnnnnnnnn*Z where values are as described in Table 15.

Table 15 — Time stamp format

Symbol	Description
YYYY	the four-digit year as measured in the Common Era.
MM	an integer between 01 and 12 corresponding to the month.
DD	an integer between 01 and 31 corresponding to the day in the month.
hh	an integer between 00 and 23 corresponding to the hour in the day.
mm	an integer between 00 and 59 corresponding to the minute in the hour.
ss	an integer between 00 and 59 corresponding to the second in the minute.
nnnnnnnnn	an integer between 000000000 and 999999999 measuring the decimal fractional second value.

NOTE: The characters '**-**', '**T**', '**:**', '**.**', and '**Z**' in the time stamp format are field separators. The '**Z**' character indicates that the time stamp is recorded in UTC (Zulu) time.

All date and time fields in the time stamp format shall be padded to the full width of the symbol using 0 characters. For example, an integer month value of '2' shall be recorded as '02' to fill the width of the *MM* symbol in the general time format.

7.8 UUID format

LTFS UUID values shall be recorded in a format compatible with OSF DCE 1.1, using 32 hexadecimal case-insensitive digits (0-9, a-f or A-F) formatted as shown. UUID values are expected to uniquely identify the LTFS Volume, as in this example:

30a91a08-daae-48d1-ae75-69804e61d2ea

8 Label Format

This section describes the content of the Label Construct. The content of the Content Area is described in Section 5.2 LTFS Constructs and in Section 9 Index Format.

8.1 Label Construct

Each partition in an LTFS Volume shall contain a Label Construct that conforms to the structure shown in Figure 16. The construct shall consist of an ANSI VOL1 Label, followed by a single file mark, followed by one record in LTFS Label format, followed by a single file mark. There shall not be any additional data trailing the end of the ANSI VOL1 Label, nor any additional data trailing the end of the LTFS Label. The Label Construct shall be recorded starting at the first logical block in the partition. Both Label constructs in an LTFS Volume shall contain identical information with the exception of the “location” field in the XML data for the LTFS Label.

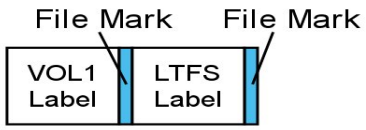


Figure 16 — Label construct

8.1.1 VOL1 Label

A VOL1 label recorded on an LTFS Volume shall always be recorded in a Label Construct as defined in Section 8.1 **Label Construct**.

The first record in a Label Construct is an ANSI VOL1 record. This record conforms to the ANSI Standard X3.27. All bytes in the VOL1 record are stored as ASCII encoded characters. The record is exactly 80 bytes in length and has the structure and content shown in Table 16.

Table 16 — VOL1 Label Construct

Offset	Length	Name	Value	Notes
0	3	label identifier	‘VOL’	
3	1	label number	‘1’	
4	6	volume identifier	<volume serial number>	Typically matches the physical cartridge label.
10	1	volume accessibility	‘L’	Accessibility limited to conformance to LTFS standard.
11	13	Reserved	all spaces	
24	13	implementation identifier	‘LTFS’	Value is left-aligned and padded with spaces to length.
37	14	owner identifier	right pad with spaces	Any printable characters. Typically reflects some user specified content oriented identification.
51	28	Reserved	all spaces	
79	1	label standard version	‘4’	

NOTE 1: Single quotation marks in the Value column above should not be recorded in the VOL1 label.

NOTE 2: All fields in the VOL1 label must contain the constant values shown in the table above. The only exceptions are the ‘volume identifier’ and ‘owner identifier’ fields. These two fields should contain user-provided values in conformance to the Notes provided.

8.1.2 LTFS Label

The LTFS Label is an XML data structure that describes information about the LTFS Volume and the LTFS Partition on which the LTFS Label is recorded. The LTFS Label shall conform to the LTFS Label XML schema provided in Annex A. The LTFS Label shall be encoded using UTF-8 NFC.

An LTFS Label recorded on an LTFS Volume shall always be recorded in an Label Construct as defined in Section 8.1 Label Construct.

A complete schema for the LTFS Label XML data structure is provided in Annex A. An example LTFS Label is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<ltfslabel version="2.5.0">
     <creator>IBM LTFS 2.5.0 - Linux - mkltfs</creator>
     <formattime>2018-10-16T18:35:47.866846222Z</formattime>
     <volumeuuid>30a91a08-daae-48d1-ae75-69804e61d2ea</volumeuuid>
     <location>
         <partition>b</partition>
     </location>
     <partitions>
         <index>a</index>
         <data>b</data>
     </partitions>
     <blocksize>524288</blocksize>
     <compression>true</compression>
</ltfslabel>
```

Every LTFS Label shall be an XML data structure that conforms to the W3C Extensible Markup Language (XML) 1.0 standard. Every LTFS Label shall have a first line that contains an XML Declaration as defined in the XML standard. The XML Declaration shall define the XML version and the encoding used for the Label.

The LTFS Label XML shall be recorded in a single logical data block and shall contain the following information:

- **ltfslabel**: this element defines the contained structure as an LTFS Label structure. The element shall have a **version** attribute that defines the format version of the LTFS Label in use. This document describes LTFS Label version 2.5.0.

  NOTE: The LTFS Label version defines the minimum version of the LTFS Format specification with which the LTFS Volume conforms. Implicitly, the LTFS Label version defines the lowest permitted version number for all LTFS Indexes written to the volume.

- **creator**: this element shall contain the necessary information to uniquely identify the writer of the LTFS volume. The value shall conform to the creator format definition shown in Section 7.2 Creator format.
- **formattime**: this element shall contain the time when the LTFS Volume was formatted. The value shall conform to the format definition shown in Section 7.7 Time stamp format.
- **volumeuuid**: this element shall contain a universally unique identifier (UUID) value that uniquely identifies the LTFS Volume to which the LTFS Label is written. The **volumeuuid** element shall conform to the format definition shown in Section 7.8 UUID format.
- **location**: shall contain a single **partition** element. The **partition** element shall specify the Partition ID for the LTFS Partition on which the Label is recorded. The Partition ID shall be a lower case ASCII character between ‘a’ and ‘z’.

- **partitions**: this element specifies the Partition IDs of the data and index partitions belonging to this LTFS volume. It shall contain exactly one **index** element for the Index Partition and exactly one **data** element for the Data Partition, formatted as shown. A partition shall exist in the LTFS Volume with a partition identifier that matches the identifier recorded in the **index** element. Similarly, a partition shall exist in the LTFS Volume with a partition identifier that matches the identifier recorded in the **data** element.
- **blocksize**: this element specifies the block size to be used when writing Data Extents to the LTFS Volume. The **blocksize** value is an integer specifying the number of 8-bit bytes that shall be written as a record when writing any full block to a Data Extent. Partial blocks may only be written to a Data Extent in conformance with the definitions provided in Section 5.2.2 Data Extent and in Section 6 Data Extents. The minimum blocksize that may be used in an LTFS Volume is 4096 8-bit bytes.

  NOTE: For general-purpose storage on data tape media the recommended blocksize is 524288 8-bit bytes.
- **compression**: this element shall contain a value conforming to the boolean format definition provided in Section 7.1 Boolean format. When the compression element is set, compression shall be enabled when writing to the LTFS Volume. When the **compression** element is unset, compression shall be disabled when writing to the LTFS Volume. The **compression** element indicates use of media-level "on-the-fly" data compression. Use of data compression on a volume is transparent to readers of the volume.

8.1.3 Managing LTFS Labels

The LTFS Label captures volume-specific values that are constant over the lifetime of the LTFS Volume. As such, the values recorded in an LTFS Label can only be set or updated at volume format time.

Implementations should handle additional unknown XML tags when they occur as children of the **ltfslabel** element. In general, such unknown tags may be ignored when mounting the LTFS Volume. This handling of unknown XML tags reduces the risk of compatibility changes when future versions of this specification are adopted. It is a strict violation of this specification to add any XML tags to the Label beyond those defined in this document.

9 Index Format

The Content Area contains zero or more Data Extents and some number of Index Constructs in any order. This section describes the content of the Index Construct. The Label Construct is described in Section 8 Label Format. Data Extents are described in Section 6 Data Extents.

9.1 Index Construct

Each Content Area in an LTFS Volume shall contain some number of Index Constructs that conform to the structure shown in Figure 17. The Index Construct shall contain a single file mark, followed by one or more records in Index format, followed by a single file mark. There shall not be any additional data trailing the end of the Index.

The contents of the Index are defined in Section 9.2 Index.

Figure 17 — Index Construct

The Index Constructs in a Content Area may be interleaved with any number of Data Extents. A complete partition shall have an Index Construct as the last construct in the Content Area, therefore there shall be at least one Index Construct per complete partition.

9.2 Index

An Index is an XML data structure that describes data files, directory information and associated metadata for files recorded on the LTFS Volume. An Index recorded on an LTFS Volume shall always be recorded in an Index Construct as defined in Section 9.1 Index Construct.

An LTFS Index is either a Full Index or an Incremental Index. A Full Index describes the state of the entire volume, i.e. all data files, directory information and associated metadata. An Incremental Index describes only changes to the volume which have occurred since the last index (Full or Incremental) was written to the volume. Full and Incremental Indexes share many of the same constructs, and the remainder of this section applies to both types unless stated otherwise.

The following rules define when Full or Incremental Indexes may be written:

- The index partition shall only contain Full Indexes, i.e. Incremental Indexes shall not be written to the index partition.
- A Full Index shall always be written to the data partition as part of the unmount processing (i.e. a cleanly unmounted volume always has a Full Index at the end of the data partition)
- An Incremental Index may be written to the data partition at any time, to store any changes to the volume contents since the last index (Full or Incremental) was written.
- A Full Index may be written to the data partition at any time, and shall represent the complete state of the volume at the time it is written.

NOTE: Prior to version 2.5.0 of this specification, all indexes were implicitly Full Indexes and were referred to simply as Indexes. The concept of an Incremental Index was introduced to reduce the space needed (and time taken) to write periodic indexes during normal operation.

Every Index shall be an XML data structure that conforms to the W3C Extensible Markup Language (XML) 1.0 standard. Every Index shall have a first line that contains an XML Declaration as defined in the

XML standard. The XML Declaration shall define the XML version and the encoding used for the Index.

An LTFS Index shall conform to the Index XML schema provided in Annex B (normative) LTFS Index XML Schema. The Index shall be encoded using UTF-8 NFC. The remainder of this section describes the content of the Index using an example XML Index.

An Index consists of a Preface section containing multiple XML elements followed by a single **directory** element. This **directory** element is referred to as the "root" **directory** element. The root **directory** element corresponds to the root of the file system recorded on the LTFS Volume.

Each **directory** element shall contain a **contents** element, which may contain zero or more **directory** elements and zero or more **file** elements. The only exception is a **directory** entry marked as **deleted** in an Incremental Index, where no **contents** element is allowed.

9.2.1 Example Full Index omitting the body

An example of a Full Index that omits the body of the **directory** element is shown in this section. The omitted section in this example is represented by the characters '...'.

```
<?xml version="1.0" encoding="UTF-8"?>
 <ltfsindex version="2.5.0">
    <creator>IBM LTFS 2.5.0 - Linux - ltfs</creator>
    <volumeuuid>30a91a08-daae-48d1-ae75-69804e61d2ea</volumeuuid>
    <generationnumber>3</generationnumber>
    <comment>A sample LTFS Index</comment>
    <updatetime>2018-10-16T19:39:57.245954278Z</updatetime>
    <location>
       <partition>a</partition>
       <startblock>6</startblock>
    </location>
    <previousgenerationlocation>
       <partition>b</partition>
       <startblock>20</startblock>
    </previousgenerationlocation>
    <allowpolicyupdate>true</allowpolicyupdate>
    <dataplacementpolicy>
       <indexpartitioncriteria>
          <size>1048576</size>
          <name>*.txt</name>
       </indexpartitioncriteria>
    </dataplacementpolicy>
    <volumelockstate>unlocked</volumelockstate>
    <highestfileuid>4</highestfileuid>
    <directory>
    ...
    </directory>
 </ltfsindex>
```

9.2.2 Example Incremental Index omitting the body

An example of an Incremental Index that omits the body of the **directory** element is shown in this section.

The omitted section in this example is represented by the characters '...'.

```
<?xml version="1.0" encoding="UTF-8"?>
 <ltfsincrementalindex version="2.5.0">
     <creator>IBM LTFS 2.5.0 - Linux - ltfs</creator>
     <volumeuuid>30a91a08-daae-48d1-ae75-69804e61d2ea</volumeuuid>
     <generationnumber>3</generationnumber>
     <comment>A sample LTFS Incremental Index</comment>
     <updatetime>2018-10-16T19:39:57.245954278Z</updatetime>
     <location>
        <partition>b</partition>
        <startblock>1632</startblock>
     </location>
     <previousgenerationlocation>
        <partition>b</partition>
        <startblock>20</startblock>
     </previousgenerationlocation>
     <previousincrementallocation>
        <partition>b</partition>
        <startblock>960</startblock>
     </previousincrementallocation>
     <volumelockstate>unlocked</volumelockstate>
     <highestfileuid>46</highestfileuid>
     <directory>
     ...
     </directory>
 </ltfsincrementalindex>
```

9.2.3 Required elements for every index

Every Index shall contain the following elements, unless otherwise noted:

- **ltfsindex** or **ltfsincrementalindex**: These elements define the contained structure as an Index structure. Every index shall contain either an **ltfsindex** element or an **ltfsincrementalindex** element. Both shall have a **version** attribute that defines the format version of the LTFS Index in use. This document describes LTFS Index version 2.5.0.

  NOTE: The LTFS Label version defines the minimum version of the LTFS Format specification with which the LTFS Volume conforms. Implicitly, the LTFS Label version defines the lowest permitted version number for all LTFS Indexes written to the volume.

  An Index update occurs when an LTFS Volume containing a current Index of version *M.N.R* is written with a new Index using a version number with a higher value for *M*. The version for any LTFS Index written to an LTFS Volume shall have an *M* value that is greater than or equal to the *M* value in the current Index. When the *M* value for the new LTFS Index equals the *M* value in the current Index, the new Index may be written in conformance to any value of *N* and *R* so long as *N* and *R* match the version of a published LTFS Format Specification.

  An Index downgrade occurs when an LTFS Volume containing a current Index of version *M.N.R* is written with a new Index using a version number with a lower value for *M*. Index downgrades are explicitly disallowed in an LTFS Volume. Further details on Index version numbering is shown in Section 2.1 Versions.

- **creator**: This element shall contain the necessary information to uniquely identify the writer of the Index. The value shall conform to the creator format definition shown in Section 7.2 Creator format.

- **volumeuuid**: This element shall contain a universally unique identifier (UUID) value that uniquely identifies the LTFS Volume to which the Index is written. The value of the **volumeuuid** element shall conform to the format definition shown in Section 7.8 UUID format. The **volumeuuid** value shall match the value of the **volumeuuid** element in the LTFS Labels written to the LTFS Volume.

- **generationnumber**: This element shall contain a non-negative integer corresponding to the generation number for the Index. The first Index on an LTFS Volume shall be generation number "1". The **generationnumber** shall conform to the definitions provided in Section 5.4.1 Generation Number.
- **updatetime**: This element shall contain the date and time when the Index was modified. The value shall conform to the format definition shown in Section 7.7 Time stamp format.
- **location**: This element shall contain a single partition element and a single **startblock** element. The **partition** element shall specify the Partition ID for the LTFS Partition on which the Index is recorded. The **startblock** element shall specify the first logical block number, within the partition, in which the Index is recorded. The **location** element is a self-pointer to the location of the Index in the LTFS Volume.
- **allowpolicyupdate**: This element shall contain a value conforming to the boolean format definition provided in Section 7.1 Boolean format. When the **allowpolicyupdate** value is set, the writer may change the content of the **dataplacementpolicy** element. When the **allowpolicyupdate** value is unset, the writer shall not change the content of the **dataplacementpolicy** element. This element is not permitted in Incremental Indexes, i.e. shall only be included as a child of an **ltfsindex** element. Additional rules for the **allowpolicyupdate** element are provided in Section 9.2.14 Data Placement Policy.
- **highestfileuid**: This element contains an integer value that is equal to the value of the largest assigned **fileuid** element in the Index. An implementation shall be able to rely on the **highestfileuid** element to determine the highest assigned **fileuid** value in the Index without traversing all **file** and **directory** elements. The valid range of values for the **highestfileuid** value is 1 through $2^{64} - 1$ with the additional special value of zero (0*x*0).

  The **highestfileuid** can be used to determine the highest integer value assigned to the **fileuid** element for all directories and files in the Index. While the **highestfileuid** value not equal to zero (0*x*0), an implementation may increment the **highestfileuid** value to create unique **fileuid** values for new directory and file entries.

  A **highestfileuid** element value of zero (0*x*0) indicates that the LTFS Volume has exhausted the contiguous range of valid values for **fileuid** elements in the Index. In this case, an implementation should use a mechanism such as traversing all **file** and **directory** elements to identify an unused and therefore unique **fileuid** value for any new **file** and **directory** elements.
- **directory**: This element corresponds to the "root" **directory** element in the Index. The content of this element is described later in this section.

9.2.4 Optional elements for every index

Every Index may contain the following elements, unless otherwise noted:

- **comment**: This element, if it exists, shall contain a valid UTF-8 encoded string value. The value of this element shall be used to store a user-provided description of this generation of the Index for the volume. The value of this element shall conform to the format definition provided in Section 7.6 String format. An Index may have at most one **comment** element. The writer of an Index may remove or replace the **comment** element when recording a new Index. The value of this element shall not exceed 64KiB in size.
- **previousgenerationlocation**: This element, if it exists, defines the back pointer for the Full Index. The **previousgenerationlocation** element shall contain a single **partition** element and a single **startblock** element. The value of the **partition** element shall specify the Partition ID for the LTFS Partition on which the back pointed Full Index is recorded. The **startblock** element shall specify the first logical block number, within the partition, in which the back pointed Full Index is recorded. If the Index does not have a back pointer there shall be no **previousgenerationlocation** element in the Index. Every Index that does have a back pointer shall have a **previousgenerationlocation**. Note that as a consequence this element is required in an Incremental Index, which will always contain a

back pointer. All data values recorded in the **previousgenerationlocation** element shall conform to the definitions provided in Section 5.4 Index Layout.

- **previousincrementallocation**: This element, if present, defines the back pointer to the most recent Incremental Index written after the most recent Full Index. The **previousincrementallocation** element shall contain a single **partition** element and a single **startblock** element. The value of the **partition** element shall specify the Partition ID for the LTFS Partition on which the back pointed Incremental Index is recorded, which must be the Data Partition since Incremental Indexes are not permitted in the Index Partition. The **startblock** element shall specify the first logical block number, within the Data Partition in which the back pointed Incremental Index is recorded. If no Incremental Index has been written since the most recent Full Index then there shall be no **previousincrementallocation** element in the Index. If an Incremental Index has been written since the most recent Full Index then this Index shall have a **previousincrementallocation**. All data values recorded in the **previousincrementallocation** element shall conform to the definitions provided in Section 5.4 Index Layout. See also Figure 7 which illustrates the relationship between **previousgenerationlocation** and **previousincrementallocation**.
- **dataplacementpolicy**: This element, if it exists, shall contain a single **indexpartitioncriteria** element. The **indexpartitioncriteria** element shall contain a single **size** element and zero or more **name** elements. The value of the **size** element shall define the maximum size of files that may be stored on the Index Partition. Each **name** element shall specify a file name pattern. The file name pattern value shall conform to the name pattern format provided in Section 7.5 Name pattern format. A description of the rules associated with the dataplacementpolicy element is provided in Section 9.2.14 Data Placement Policy. This element may exist in a Full Index but shall not exist in an Incremental Index.
- **volumelockstate**: This element, if it exists, indicates the state of volume advisory locking for the volume. The following values are defined:

unlocked	The volume may be modified
locked	The volume shall not be modified other than to change the **volumelockstate**
permlocked	The volume is permanently locked and shall not be modified in any way

If a volume is in the **unlocked** state, it may be modified either to the **locked** state or to the **permlocked** state.

If a volume is in the **locked** state, it may be modified either to the **unlocked** state or to the **permlocked** state.

If a volume is in the **permlocked** state, it may be reformatted to enable re-use of the cartridge; however no other write / update operations are permitted.

If this element does not exist then the volume is implicitly treated as unlocked.

Refer to Section 9.2.19 for more information.

9.2.5 Example Full Index that omits the Preface section

An example Full Index that omits the Preface section of the Index is shown in this section. The omitted section in this example is represented by the characters '...'. This example shows the root **directory** element for the Index.

```
<?xml version="1.0" encoding="UTF-8"?>
 <ltfsindex version="2.5.0">
    ...
```

```
    <directory>
       <fileuid>1</fileuid>
       <name>LTFS Volume Name</name>
       <creationtime>2018-10-17T19:39:50.715656751Z</creationtime>
       <changetime>2018-10-17T19:39:55.231540960Z</changetime>
       <modifytime>2018-10-17T19:39:55.231540960Z</modifytime>
       <accesstime>2018-10-17T19:39:50.715656751Z</accesstime>
       <backuptime>2018-10-17T19:39:50.715656751Z</backuptime>
       <contents>
          <directory>
             <fileuid>2</fileuid>
             <name>directory1</name>
             <creationtime>2018-10-17T19:39:50.740812831Z</creationtime>
             <changetime>2018-10-17T19:39:56.238128620Z</changetime>
             <modifytime>2018-10-17T19:39:54.228983707Z</modifytime>
             <accesstime>2018-10-17T19:39:50.740812831Z</accesstime>
             <backuptime>2018-10-17T19:39:50.740812831Z</backuptime>
             <readonly>false</readonly>
             <contents>
                <directory>
                   <fileuid>3</fileuid>
                   <name>subdir1</name>
                   <readonly>false</readonly>
                   <creationtime>2018-10-17T19:39:54.228983707Z</creationtime>
                   <changetime>2018-10-17T19:39:54.228983707Z</changetime>
                   <modifytime>2018-10-17T19:39:54.228983707Z</modifytime>
                   <accesstime>2018-10-17T19:39:54.228983707Z</accesstime>
                   <backuptime>2018-10-17T19:39:54.228983707Z</backuptime>
                </directory>
             </contents>
          </directory>
          <file>
             <fileuid>4</fileuid>
             <name>testfile.txt</name>
             <length>5</length>
             <creationtime>2018-10-17T19:39:51.744583047Z</creationtime>
             <changetime>2018-10-17T19:39:57.245291730Z</changetime>
             <modifytime>2018-10-17T19:39:57.245291730Z</modifytime>
             <accesstime>2018-10-17T19:39:57.240774456Z</accesstime>
             <backuptime>2018-10-17T20:21:45.424385077Z</backuptime>
             <readonly>true</readonly>
             <extendedattributes>
             </extendedattributes>
             <extentinfo>
                <extent>
                   <partition>a</partition>
                   <startblock>4</startblock>
                   <byteoffset>0</byteoffset>
                   <bytecount>5</bytecount>
                   <fileoffset>0</fileoffset>
                </extent>
             </extentinfo>
          </file>
       </contents>
    </directory>
 </ltfsindex>
```

9.2.6 Required directory elements for a Full Index

An Index shall have exactly one **directory** element recorded as a child of the **ltfsindex** element in the Index. The **directory** element recorded as a child of the **ltfsindex** element in the Index shall represent the root of the filesystem on the LTFS Volume.

Every **directory** element (at any level) shall contain the following information:

- **fileuid**: This element shall contain an integer value that is a unique identifier with respect to directories and files in the Index. The valid range of values for the **fileuid** value is 1 through $2^{64} - 1$.

  An example of how to calculate this unique value is provided in the description of **highestfileuid** above. The **directory** element corresponding to the root of the filesystem shall have a **fileuid** value of one (0*x*1).

  name: This element shall contain the name of the directory. A directory name shall conform to the format specified in Section 7.4 Name format. The value of the name element for the root directory element in an Index shall be used to store the name of the LTFS Volume.

- **creationtime**: This element shall contain the date and time when the directory was created in the LTFS Volume. The value shall conform to the format definition shown in Section 7.7 Time stamp format.

- **changetime**: This element shall contain the date and time when the extended attributes or readonly element for the directory was last altered. The value shall conform to the format definition shown in Section 7.7 Time stamp format.

- **modifytime**: This element shall contain the date and time when the content of the directory was most recently altered. The value shall conform to the format definition shown in Section 7.7 Time stamp format.

- **accesstime**: This element may contain the date and time when the content of the directory was last read. Implementators of the LTFS Format may choose to avoid or otherwise minimize recording Index updates that only change the **accesstime** element. The value shall conform to the format definition shown in Section 7.7 Time stamp format.

- **backuptime**: This element may contain the date and time when the content of the directory was last archived or backed up. If the directory has never been archived or backed up this element shall contain a value equal to the value of the **createtime** element. The value shall conform to the format definition shown in Section 7.7 Time stamp format.

- **readonly**: This element shall contain a value conforming to the boolean format definition provided in Section 7.1 Boolean format. When the **readonly** element is set, the directory shall not be modified by any writer. When the **readonly** element is unset, the directory may be modified by any writer. The following operations are considered to be modifications to a directory:
  - adding a child file or directory
  - removing a child file or directory, and
  - any change to the **extendedattributes** element.

- **contents**: This element shall contain zero or more **directory** elements and zero or more **file** elements. The elements contained in the **contents** element are children of the directory.

9.2.7 Optional directory elements for a Full Index

Every **directory** element may contain the following elements:

- **extendedattributes**: This element, if it exists, may contain zero or more **xattr** elements. The **xattr** elements are described in Section 9.2.10 extendedattributes elements. A **directory** element may have zero or one **extendedattributes** elements.

9.2.8 Required file elements for a Full Index

Every **file** element shall contain the following information:

- **fileuid**: This element shall contain an integer value that is a unique identifier with respect to directories and files in the Index. The valid range of values for the **fileuid** value is 2 through $2^{64} - 1$. An example of how to calculate this unique value is provided in the description of **highestfileuid** above.

NOTE: The value of the 'fileuid' element for the root directory is one (0x01) as defined in Section 9.2.5

All 'fileuid' elements shall be unique in the index therefore no file may have a 'fileuid' less than 2.

- **name**: This element shall contain the name of the file. A file name shall conform to the format specified in Section 7.4 Name format.
- **length**: for file elements containing an **extentinfo** element or file elements describing a regular file with no **extentinfo** element (zero length or sparse files), the **length** element shall contain the integer length of the file. The length is measured in bytes. For **file** elements containing a **symlink** element, the length element shall contain the integer length of the **symlink** target path.
- **creationtime**: This element shall contain the date and time when the file was created in the LTFS Volume. The value shall conform to the format definition shown in Section 7.7 Time stamp format.
- **changetime**: This element shall contain the date and time when the extended attributes or readonly element for the file was last altered. The value shall conform to the format definition shown in Section 7.7 Time stamp format.
- **modifytime**: This element shall contain the date and time when the content of the file was most recently altered. The value shall conform to the format definition shown in Section 7.7 Time stamp format.
- **accesstime**: This element may contain the date and time when the content of the file was last read. Implementers of the LTFS Format may choose to avoid or otherwise minimize recording Index updates that only change the **accesstime** element. The value shall conform to the format definition shown in Section 7.7 Time stamp format.
- **backuptime**: This element may contain the date and time when the content of the file was last archived or backed up. If the file has never been archived or backed up, this element shall contain a value equal to the value of the **createtime** element. The value shall conform to the format definition shown in Section 7.7 Time stamp format.
- **readonly**: This element shall contain a value conforming to the boolean format definition provided in Section 7.1 Boolean format. When the **readonly** element is set, the file shall not be modified by any writer. When the **readonly** element is unset, the file may be modified by any writer. The **readonly** element is ignored for file elements containing a symlink element.

9.2.9 Optional file elements for a Full Index

Every **file** element may contain the following elements:

- **extendedattributes**: This element, if it exists, may contain zero or more **xattr** elements. The xattr elements are described in Section 9.2.10 extendedattributes elements. A **file** element may have zero or one **extendedattributes** elements.
- **extentinfo**: This element, if it exists, may contain zero or more **extent** elements. A **file** element may have zero or one **extentinfo** elements, however a file element shall not have both an **extentinfo** element and a **symlink** element .

  Every **extent** element shall describe the location where a file extent is recorded in the LTFS Volume. Every **extent** element shall contain one **partition** element, one **startblock** element, one **byteoffset** element, one **bytecount** element, and one **fileoffset** element. The values recorded in elements contained by the **extentinfo** element shall conform to the definitions provided in Section 5.2.2 Data Extent and in Section 6 Data Extents. The **partition** element shall contain the Partition ID corresponding to the LTFS partition in which the Data Extent is recorded. The **startblock** element shall specify the first logical block number, within the partition, in which the Data Extent is recorded. The **byteoffset** element shall specify the offset into the start block within the Data Extent at which the valid data for the extent is recorded. The **bytecount** element shall specify the number of bytes that comprise the extent. The **fileoffset** element shall specify the offset into the file where the data stored in this Data Extent starts.

  The order of **extent** elements within an **extentinfo** element is not significant. Implementors are

encouraged to record **extentinfo** in the same order that the extents occur in the file. The definition of how extent values are determined and used is provided in Section 6 Data Extents and in Section 6.1 Extent Lists.

- **openforwrite**: This element, if it exists, shall contain a value conforming to the boolean format definition provided in Section 7.1 Boolean format. When the **openforwrite** element exists and has the value **true**, the corresponding file was open for writing at the time that the index was written and so may not be complete. This information may be made available to the user by an application attempting to roll back a volume, for example to inform the user's choice of rollback points.

  If the **openforwrite** element exists and has the value **false**, or if it does not exist, then there is no indication of whether or not the file was open for writing.

  An application claiming conformance to version 2.4 or later of this specification shall include the **openforwrite** element with the value **true** for all files open for writing when the index was written.

NOTE 1: It is recommended that this element should only be included for files that are known to be in the open state, i.e. not included if the value would be **false**.

NOTE 2: In normal usage the index (in both partitions) written on a clean unmount will not have the **openforwrite** element set to **true**, as all files should be closed as part of the unmount processing.

- **symlink**: This element, if it exists, shall contain either the fully qualified path from the root of the file system tree to the target file, or shall contain a relative path to the target file. Path strings shall be stored using the Unix-style forward slash as the path delimiter. The path shall conform to the format specified in Section 7.4 Name format. A **file** element may have zero or one **symlink** elements, however a **file** element shall not have both an **extentinfo** element and a **symlink** element.

NOTE: It is possible that an older implementation of LTFS could create a tape that violates the mutual exclusivity requirement for extentinfo and symlink elements. In this case, the LTFS volume will not conform to this specification; it is recommended that an LTFS implementation encountering such a volume perform a recovery action before mounting or using the volume.

9.2.10 extendedattributes elements

All **directory** and **file** elements in an Index may specify zero or more extended attributes. These extended attributes are recorded as **xattr** elements in the **extendedattributes** element for the **directory** or **file**.

An example **directory** element is shown in the following paragraph, with three extended attributes recorded. The empty_xattr and document_name extended attributes in this example both record string values. The binary_xattr attribute is an example of storing a binary extended attribute value. This example omits parts of the Index outside of the directory. The omitted sections in this example are represented by the characters "...".

```
...
<directory>
   <fileuid>2</fileuid>
   <name>directory1</name>
   <creationtime>2013-01-28T19:39:50.740812831Z</creationtime>
   <changetime>2013-01-28T19:39:56.238128620Z</changetime>
   <modifytime>2013-01-28T19:39:54.228983707Z</modifytime>
   <accesstime>2013-01-28T19:39:50.740812831Z</accesstime>
   <backuptime>2013-01-28T19:39:50.740812831Z</backuptime>
   <extendedattributes>
      <xattr>
         <key>binary_xattr</key>
         <value  type="base64">/42n2QaEWDSX+g==</value>
      </xattr>
      <xattr>
         <key>empty_xattr</key>
         <value/>
```

```
            </xattr>
            <xattr>
               <key>document_name</key>
               <value type="text">LTFS Format Specification</value>
            </xattr>
         </extendedattributes>
         <contents>
         </contents>
      </directory>
      ...
```

Each **extendedattributes** element may contain zero or more **xattr** elements.

Each **xattr** element shall contain one **key** element and one **value** element. The **key** element shall contain the name of the extended attribute. The name of the extended attribute shall conform to the format specified in Section 7.4 Name format. Extended attribute names shall be unique within any single **extendedattributes** element. The **value** element shall contain the value of the extended attribute. The **value** element may have a **type** attribute that defines the type of the extended attribute value. If the **type** attribute is omitted then the type for the extended attribute value shall be "text". The value of the extended attribute shall conform to the format specified in Section 7.3 Extended attribute value format.

All extended attribute names that match the prefix "ltfs" with any capitalization are reserved for use by the LTFS Format. (That is, any name starting with a case-insensitive match for the letters "ltfs" are reserved.) Any writer of an LTFS Volume shall only use reserved extended attribute names to store extended attribute values in conformance with the reserved extended attribute definitions shown in Annex C.

9.2.11 Required and Optional elements for Incremental Indexes

Because Incremental Indexes record only changes to the volume contents, the preceding sections may not apply in their entirety; however the following rules do apply:

- Newly created directories shall follow the rules described in 9.2.6 Required directory elements for a Full Index and 9.2.7 Optional directory elements for a Full Index.
- Newly created files shall follow the rules described in 9.2.8 Required file elements for a Full Index and 9.2.9 Optional file elements for a Full Index.
- Modified directories or files shall follow the same rules as for newly created directories or files except for the list of required elements. The **name** and **fileuid** elements are required and shall be recorded; any elements which have changed since the last index shall also be recorded. Elements which have not changed since the last index are not required to be recorded, but may be included if desired to make the implementation more straightforward. Note that if the **extentinfo** or **extendedattributes** elements have changed for a file or directory, they shall be included in an Incremental Index in their entirety.
- Deleted files shall be denoted using a special **file** element **deleted**. If this element is present in a file entry, then the **name** and **deleted** elements are the only elements that shall appear in the Incremental Index entry for that file. Every file deleted since the last index shall be included in the Incremental Index and shall include the **deleted** element. The only exception to this rule is where a parent directory has also been deleted (and so includes the **deleted** element); in this case the file is implicitly assumed to have been deleted or moved.
- Deleted directories shall also be denoted using the optional **deleted** element. If this element is present in a directory entry, then the **name** and **deleted** elements are the only elements that shall appear in the Incremental Index entry for that directory. Every directory deleted since the last index shall be included in the Incremental Index and shall include the **deleted** element. The only exception to this rule is where a parent directory has also been deleted (and so includes the **deleted** element); in this case the child directory is implicitly assumed to have been deleted or moved.

NOTE: Normal file system operation requires that any children of a directory must previously have been deleted or moved/renamed as a prerequisite for the deletion of the directory.

- Files or directories that have been moved or renamed shall be reflected in an Incremental Index using the **deleted** element for the old name and the insertion of the file or directory with its new name. A directory inserted due to a move or rename must include all of the information about the directory itself, along with all children (at any level) that have been relocated with it. In other words, the inserted directory must record all of the information about the entire directory subtree that was moved or renamed.
- The **deleted** element shall not appear in a Full Index.
- Refer to Annex H for more information on handling Incremental Indexes.

9.2.12 Example Incremental Index that omits the Preface section

An example Incremental Index that omits the Preface section of the index is shown in this section. The omitted section in this example is represented by the characters '…'. This example illustrates the case where a new file samplefile.txt has been created in the top level directory, and both an empty directory subdir1 and a file named testfile.txt have been deleted. The modifytime timestamps for the affected parent directories are updated.

```
<?xml version="1.0" encoding="UTF-8"?>
 <ltfsincrementalindex version="2.5.0">
    ...
    <directory>
       <fileuid>1</fileuid>
       <name>LTFS Volume Name</name>
       <modifytime>2018-10-02T19:39:54.228983707Z</modifytime>
       <changetime>2018-10-02T19:39:54.228983707Z</changetime>
       <contents>
          <directory>
             <fileuid>2</fileuid>
             <name>directory1</name>
             <modifytime>2018-10-02T19:39:54.228984707Z</modifytime>
             <changetime>2018-10-02T19:39:54.228984707Z</changetime>
             <contents>
                <directory>
                   <name>subdir1</name>
                   <deleted/>
                </directory>
             </contents>
          </directory>
          <file>
             <name>testfile.txt</name>
             <deleted/>
          </file>
          <file>
             <fileuid>5</fileuid>
             <name>samplefile.txt</name>
             <length>256</length>
             <creationtime>2018-10-02T19:39:54.228983701Z</creationtime>
             <changetime>2018-10-02T19:39:54.228983701Z</changetime>
             <modifytime>2018-10-02T19:39:54.228983701Z</modifytime>
             <accesstime>2018-10-02T19:39:54.228983701Z</accesstime>
             <backuptime>2018-10-02T19:39:54.228983701Z</backuptime>
             <readonly>false</readonly>
             <extendedattributes>
             </extendedattributes>
             <extentinfo>
                <extent>
                   <partition>b</partition>
                   <startblock>18</startblock>
                   <byteoffset>0</byteoffset>
                   <bytecount>256</bytecount>
```

```
                    <fileoffset>0</fileoffset>
                </extent>
            </extentinfo>
        </file>
    </contents>
</directory>
</ltfsincrementalindex>
```

9.2.13 Managing LTFS Indexes

A Full Index is a snapshot representation of the entire content of the LTFS Volume at a given point in time. An Incremental Index is a snapshot representation of changes to the volume since the previous index (Full or Incremental) was written. Incremental Indexes may be written as needed to save the current state of the volume, but shall always be followed by a Full Index generated during unmount processing or as needed. An implementation claiming compliance with v2.5.0 or later of this specification shall be able to recognize and process Incremental Indexes, but is not required to write them. An implementation processing a consistent volume for mount will not normally need to know whether Incremental Indexes have been written in previous mounts.

NOTE: This should minimize backward compatibility issues, allowing an older implementation to mount a volume containing Incremental Indexes without understanding them. Checking and rolling back of a volume containing Incremental Indexes will be more complex; it is here that backwards compatibility will be an issue since an older implementation will not be able to walk the full back pointer chain. Because an older implementation will not recognize incremental indexes, it will ignore them and use the Full Index back pointer chain for rollback. Rollback can be performed, but it will be possible to roll back only to a Full Index (not to any Incremental Indexes). Using an older implementation to recover a volume that is in an inconsistent state and that contains Incremental Indexes may fail or lead to data loss if the most recent index on the volume is an Incremental Index; for more details, refer to Section H.2 Backwards Compatibility.

An implementation should allow the user to specify the interval between Full Indexes, i.e. how many Incremental Indexes may be written before a Full Index is required. For example if this value were set to 5, then after five Incremental Indexes had been written to tape, the next index written would be a Full Index. If the interval were set to 0 then no Incremental Indexes would be written and the behavior would be unchanged from versions of this specification prior to v2.5.0. An implementation may choose to write a Full Index at any time.

NOTE: It is recommended that an implementation should set a limit on this interval, because of the increased complexity and time required to rebuild a full index from a sequence of incremental indexes. A maximum value in the range 5-10 may be suitable.

Implementations should handle additional unknown XML tags when they occur as children of the **ltfsindex**, **ltfsincrementalindex**, **directory**, and **file** elements. These additional tags shall be preserved when a new generation of the Index is written to the LTFS Volume. This handling of unknown XML tags reduces the risk of compatibility changes when future versions of this specification are adopted. It is a strict violation of this specification to add any XML tags to the Index beyond those defined in this document.

9.2.14 Data Placement Policy

A Full Index may specify a Data Placement Policy. This policy defines when the Data Extents for a file may be placed on the Index Partition. A Data Placement Policy specifies the conditions under which it is allowed to place Data Extents on the Index Partition. An Incremental Index shall not specify a Data Placement Policy.

An example Full Index that shows the elements that define the Data Placement Policy for an LTFS Volume is shown in this section. This example omits part of the Preface section and the root directory element. The omitted sections in this example are represented by the characters '...'.

```
<?xml version="1.0" encoding="UTF-8"?>
 <ltfsindex version="2.5.0">
```

```
    ...
    <allowpolicyupdate>true</allowpolicyupdate>
    <dataplacementpolicy>
         <indexpartitioncriteria>
           <size>1048576</size>
           <name>*.txt</name>
           <name>*.bin</name>
         </indexpartitioncriteria>
    </dataplacementpolicy>
    <directory>
    ...
    </directory>
</ltfsindex>
```

The Data Placement Policy for an LTFS Volume shall be defined in a **dataplacementpolicy** element in a Full Index. A Full Index may contain zero or one **dataplacementpolicy** elements.

Every **dataplacementpolicy** element shall contain exactly one **indexpartitioncriteria** element. This means that the **dataplacementpolicy** constructs <dataplacementpolicy/> and <dataplacementpolicy></dataplacementpolicy> are explicitly disallowed.

Every **indexpartitioncriteria** element shall contain exactly one **size** element. The **size** element shall define the maximum file size for the Data Placement Policy.

Every **indexpartitioncriteria** element may contain zero or more **name** elements. The value of each **name** element shall define a Filename Pattern for the Data Placement Policy. The Filename Pattern value shall conform to the format defined in Section 7.5 Name pattern format.

9.2.15 Data Placement Policy Alteration

An LTFS Volume shall have an associated Allow Policy Update value. The current Allow Policy Update value for an LTFS Volume shall be defined in the most recent Full Index as described in Section 9.2.14 Data Placement Policy.

This section describes the conditions under which the Data Placement Policy and Allow Policy Update values may be altered.

9.2.16 Allow Policy Update is set

If the current Allow Policy Update value is set, as defined in Section 9.2.14 Data Placement Policy, a writer may record a Full Index that indicates the Allow Policy Update value is set or unset.

If the current Allow Policy Update value is set, as defined in Section 9.2.14 Data Placement Policy, a writer may record a Full Index with the same **dataplacementpolicy** values recorded in the previous generation of the Index.

If the current Allow Policy Update value is set, as defined in Section 9.2.14 Data Placement Policy, a writer may record a Full Index with **dataplacementpolicy** values that differ from the **dataplacementpolicy** values recorded in the previous generation of the Index.

If the current Allow Policy Update value is set, as defined in Section 9.2.14 Data Placement Policy, a writer may record a Full Index without any **dataplacementpolicy** element.

9.2.17 Allow Policy Update is unset

If the current Allow Policy Update value is unset, as defined in Section 9.2.14 Data Placement Policy, a writer shall only record a Full Index that indicates the Allow Policy Update is unset.

If the current Allow Policy Update value is unset, as defined in Section 9.2.14 Data Placement Policy, a writer shall only record a Full Index without a **dataplacementpolicy** element when the previous generation of the Index does not contain a **dataplacementpolicy** element.

If the current Allow Policy Update value is unset, as defined in Section 9.2.14 Data Placement Policy, a writer shall only record a Full Index with **dataplacementpolicy** values when those values exactly match

the **dataplacementpolicy** values recorded in the previous generation of the Index.

9.2.18 Data Placement Policy Application

An LTFS Volume may have an associated Data Placement Policy. The current Data Placement Policy for an LTFS Volume shall be defined in the current Full Index as described in Section 9.2.14 Data Placement Policy. This section describes how the current Data Placement Policy and current Allow Policy Update value shall affect the valid placement options for Data Extents when adding files to an LTFS Volume.

The Data Placement Policy defines criteria controlling the conditions under which Data Extents may be recorded to the Index Partition. The current Data Placement Policy only affects the placement of Data Extents for new files written to the LTFS Volume. The Data Placement Policy has no impact on Data Extents already written to the LTFS Volume. Similarly, the Data Placement Policy does not imply any constraint on Data Extents previously written to the LTFS Volume.

The Data Placement Policy in use for an LTFS Volume does not require that Data Extents conforming to the policy be written to the Index Partition. A Data Placement Policy only defines the conditions under which it is valid to write Data Extents to the Index Partition. When the Data Placement Policy in use does not allow a Data Extent to be written to the Index Partition the Data Extent shall be written to the Data Partition. Any Data Extent may be written to the Data Partition regardless of the Data Placement Policy in use.

Any LTFS Volume without a defined Data Placement Policy, as described in Section 9.2.14 Data Placement Policy, shall have a NULL Data Placement Policy.

A NULL Data Placement Policy shall mean that no criteria exist to control the conditions under which Data Extents may be recorded to the Index Partition. When a NULL Data Placement Policy is in effect, any Data Extent may be written to the Index Partition. In general, it is recommended that implementations should avoid use of NULL Data Placement Policies.

A Data Placement Policy other than the NULL policy shall define the criteria under which the Data Extents for a new file may be written to the Index Partition.

A non-NULL Data Placement Policy shall define a maximum file size for the policy. The maximum file size may be “0” or any positive integer.

A non-NULL Data Placement Policy may define zero or more Filename Pattern values for the policy. The Filename Pattern values shall be defined and interpreted as file name patterns conforming to the format defined in Section 7.5 Name pattern format.

A non-NULL Data Placement Policy shall “match” the Data Extents being recorded to an LTFS Volume if and only if all of the following conditions are met:

- the size of the file being recorded is smaller than the maximum file size for the Data Placement Policy in effect, and
- the file name of the file being recorded matches any of the file name patterns defined in the Data Placement Policy. The rules for matching file name patterns to file names are provided in Section 7.5 Name pattern format.

NOTE: Files with a size of 0 bytes have no Data Extents recorded anywhere in the volume. Therefore, a Data Placement Policy with size value of “0” indicates that no file shall have Data Extents stored on the Index Partition.

As described in Section 9.2 Index, every Full Index shall contain a boolean **allowpolicyupdate** element corresponding to the Allow Policy Update value for the Index. When Allow Policy Update is unset, a writer shall not modify an LTFS Volume unless the modification conforms with the Data Placement Policy defined for the Index. Any writer unable to comply with the current Data Placement Policy shall leave the LTFS Volume unchanged.

Writers are encouraged to comply with the current Data Placement Policy at all times. However, when Allow Policy Update is set, a writer is permitted to violate the Data Placement Policy. Violating the policy in this case is equivalent to changing the Policy, modifying the Volume, then changing the Policy back to the original Policy.

NOTE: It is always valid to write a non-empty Data Extent to the Data Partition. This results from the Data Placement Policy and Allow Policy Update values defining when it is permitted to write Data Extents to the Index Partition rather than these values defining when it is required that Data Extents be written to the Index Partition.

9.2.19 Volume Advisory Locking

Although most tape cartridges incorporate some form of physical write protect mechanism, it is also useful for a software application to be able to mark an LTFS volume as write-protected. This is achieved through the Volume Advisory Locking mechanism, whereby the application modifies the index to indicate the locked state of the volume. Refer to Section 9.2.4 for details of permitted operations on a locked volume.

It is recommended that a volume which has been locked should be mounted as Read Only to prevent inadvertent modification, and to indicate to the user that the volume is in a locked state.

An implementation which claims to support version 2.3.0 or later of this specification shall support this Volume Advisory Locking mechanism and shall honor the locked state of the volume. It is important to note that if a locked volume is mounted by an application that complies with an earlier version of this format specification, the application will be unaware of the Volume Advisory Locking mechanism and so will permit changes. Also any different application can also overwrite or modify the volume; the Volume Advisory Locking mechanism is not intended to guard against all possible modifications. To guarantee that no further changes can be made to the volume, it is necessary to use the physical write protect mechanism of the cartridge.

10 Medium Auxiliary Memory

An LTFS Volume may use standard Medium Auxiliary Memory (MAM) to store auxiliary information with the volume to improve the efficiency of LTFS Index retrieval and to aid the identification and management of an LTFS Volume. Values stored in the MAM are stored on the volume in non-volatile storage as MAM attributes. Use of these attributes can enhance performance of an implementation but are not required for compliance to the LTFS Format Specification. That is, an LTFS Volume may still be correctly read and written if the MAM attributes become inaccessible or are not updated.

For each partition, LTFS stores a standardized Volume Coherency Information (VCI) value in a MAM attribute. This attribute contains a standardized value known as the Volume Change Reference (VCR), together with the Index generation number for the current Index and the on-media location of the current Index. These values can be used to determine whether a partition is complete and to verify volume consistency without requiring that the Index be read from both partitions. This allows an implementation to avoid the cost of seeking to the end of both partitions when verifying the consistency of an LTFS Volume.

For performance reasons, it is strongly recommended that LTFS implementers use the MAM attributes as described in Section 10.3 Use of Volume Coherency Information for LTFS if such usage is supported by the underlying storage technology.

Standard MAM attributes can be used to identify the volume as containing LTFS format, and it is strongly recommended that LTFS implementers populate the attributes described in Section 10.4 Use of Host-type Attributes for LTFS. Note that some of the attributes are mandatory for implementations which claim compliance to revision 2.2.0 or later of the LTFS format specification and where MAM attributes are supported by the underlying storage technology.

NOTE: For consistency with the referenced specifications, throughout Section 10 Medium Auxiliary Memory, the word *Volume* is used to refer to a data storage medium (e.g., a tape cartridge). The words *LTFS Volume* is used when referencing an 'LTFS Volume' as defined in Section 4.1.20

LTFS Volume and throughout this document.

10.1 Volume Change Reference

Volume Change Reference (VCR) is a non-repeating, unique value associated with a volume coherency point. This section contains a partial description of the VCR (for informational purposes). See the T10/SSC4 Standard for a complete description of the VCR.

The VCR attribute indicates changes in the state of the medium related to logical objects or format specific symbols of the currently mounted volume. There is one value for the volume change reference. The VCR attribute for each partition shall use the same single VCR value. The VCR attribute value shall:

- be written to non-volatile medium auxiliary memory before the change on medium is valid for reading, and
- change in a non-repeating fashion (i.e., never repeat for the life of the volume).

The VCR attribute value shall change when:

- the first logical object for each mount is written on the medium in any partition;
- the first logical object is written after GOOD status has been returned for a READ ATTRIBUTE command with the SERVICE ACTION field set to ATTRIBUTE VALUES (i.e., 0x00) and the FIRST ATTRIBUTE IDENTIFIER field set to VOLUME CHANGE REFERENCE (i.e., 0x0009);
- any logical object on the medium (i.e., in any partition) is overwritten; or
- the medium is formatted.

The VCR attribute may change at other times when the contents on the medium change. The VCR attribute should not change if the logical objects on the medium do not change.

A binary value of all zeros (e.g., 0x0000) in the VCR attribute indicates that the medium has not had any

logical objects written to it (i.e., the volume is blank and has never been written to) or the value is unknown. A binary value of all ones (e.g., 0xFFFF) in the VCR attribute indicates that the VCR attribute has overflowed and is therefore unreliable. In this situation, the VCR value shall not be used.

10.2 Volume Coherency Information

The Volume Coherency Information (VCI) attribute contains information used to maintain coherency of information for a volume. The VCI has six fields as listed in Table 17. There shall be one VCI attribute for each LTFS Partition that is part of an LTFS Volume. The correspondence between LTFS nomenclature and T10/SSC-4 nomenclature is shown in Table 17.

Table 17 shows a partial listing of the Volume Coherency Information attribute (for informational purposes). See the T10/SSC-4 Standard for a complete description of the Volume Coherency Information attribute.

Table 17 — Volume Coherency Information

LTFS Name	T10 SSC-4 Name
VCR Length	VOLUME CHANGE REFERENCE VALUE LENGTH
VCR	VOLUME CHANGE REFERENCE VALUE
generation number	VOLUME COHERENCY COUNT
block number	VOLUME COHERENCY SET IDENTIFIER
Application Client Specific Information Length	APPLICATION CLIENT SPECIFIC INFORMATION LENGTH
Application Client Specific Information	APPLICATION CLIENT SPECIFIC INFORMATION

Notes for Table 17:

1. VCR Length: this field contains the length of the VCR field. The VCR Length field is a one-byte field.
2. VCR: this field contains the value returned in the VCR attribute after all information for which coherency is desired was written to the volume. The length of this field is specified by the value of the VCR Length field.
3. generation number: this field contains the generation number of the LTFS Index that is pointed to by the block number field. The generation number field is an 8-byte field. The value stored in this field shall be a big-endian binary integer value.
4. block number: this field contains the logical block number of the LTFS Index on this partition for which coherency is desired. Typically coherency is desired for the most recently written LTFS Index. This field and the partition ID of this partition comprise the position of the LTFS Index on the media. A value of zero is invalid. The block number field is an 8-byte field.
5. Application Client Specific Information Length: this field contains the length of the Application Client Specific Information field. The Application Client Specific Information Length field is a two-byte field.
6. Application Client Specific Information: this field contains information the application client associates with this coherency set. The length of this field is specified by the value of the Application Client Specific Information Length field.

10.3 Use of Volume Coherency Information for LTFS

Use of the Volume Coherency Information (VCI) attribute with the LTFS format is optional, but it is recommended to improve performance. If the VCI attribute is stored for an LTFS Partition, it shall be used as described in this section.

The VCI attribute for each volume partition contains the Application Client Specific Information (ACSI) for the LTFS Partition stored on the volume partition. The ACSI for LTFS shall be formatted as shown in Table 18. All offsets and lengths are measured in bytes.

Table 18 — ACSI format for LTFS

Offset	Length	Value	Notes
0	4	'LTFS'	
4	1	0x00	string terminator (binary)
5	36	<volume UUID>	as defined in Section 7.8 UUID format
41	1	0x00	string terminator (binary)
42	1	0x01	version number (binary)

NOTE: Single quotation marks in the 'Value' column shall not be recorded in the Application Client Specific Information.

The first 43 bytes of the Application Client Specific Information will retain their current meaning in all future versions of the LTFS Format. A future version of the LTFS Format may define additional content to be appended to the Application Client Specific Information, in which case the version number field will be incremented.

NOTE: The version number stored at offset 42 has been incremented from ***0x0*** in IBM LTFS Format Specification version 1.0 to *0x1* for LTFS Format Specification version 2.0.0. This increment allows identification of LTFS Volumes created with incorrect MAM values by an implementation of the IBM LTFS Format Specification version 1.0.

An application may write the VCI attribute for an LTFS Partition at any time when the partition is complete. The attribute shall contain the VCR of the cartridge and the generation number of the last LTFS Index on the partition, with both values determined at the time the attribute is written. When writing the VCI attribute for any LTFS Partition, an application should write the VCI attribute for all complete partitions. Implementations of the LTFS Format Specification should update the VCI attribute for all complete partitions immediately after fully writing an Index Construct to any partition. The recommended order of operations is:

1. Write an Index Construct to a partition.
2. Ensure that all pending write requests are flushed to the medium. The procedure for doing this may depend on the underlying storage technology.
3. Read the VCR attribute immediately (before issuing any additional write requests to the medium).
4. If the VCR attribute value is valid (i.e., does not contain a binary value of all ones or all zeros), compute and write the VCI attributes containing the read VCR value for all complete partitions.

A VCR instance in a VCI attribute is up-to-date if it equals the VCR value of the cartridge. Any LTFS Partition with a corresponding VCI attribute that contains an up-to-date VCR instance is complete. If all partitions in an LTFS Volume have VCI attributes containing up-to-date VCR instances, the attribute with the highest generation number determines the block position of the current Index for the LTFS Volume. This allows an implementation to determine the state of an LTFS Volume quickly by reading that single LTFS Index.

If any partition in an LTFS Volume has a VCI attribute containing a VCR instance which is not up-to-date, that partition is not guaranteed to be complete. In this case, the consistency of the LTFS Volume cannot be determined from the values in the VCI attributes for each partition. For example, the following sequence of operations results in exactly one partition having a VCI attribute containing an up-to-date VCR instance but the LTFS Volume is not consistent:

1. An implementation writes an Index Construct to partition 'a', then writes the VCI attribute for partition 'a'.
2. The implementation appends a Data Extent to partition 'a'. The VCI attribute for partition 'a' now contains an out-of-date VCR instance.
3. The implementation Writes an Index Construct to partition 'b', then writes the VCI attribute for partition 'b'.

In this case, the current Index for the LTFS Volume cannot be identified without reading Indexes from both partitions and comparing their generation numbers.

10.4 Use of Host-type Attributes for LTFS

The T10 technical committee of INCITS owns the specification for MAM attributes (published in the SCSI Primary Commands standard SPC-4), and these attributes include a category known as Host-type Attributes intended to provide host-settable information describing the volume. For full details of these attributes refer to the T10/SPC-4 Standard.

The relevant attributes are shown in Table 19. The "Support" column indicates whether implementations which claim compliance to revision 2.4.0 or later of the LTFS format specification should support (O – optional) or shall support (M- mandatory) the corresponding attribute.

Table 19 — Relevant Host-type Attributes for LTFS

Attribute Name	Identifier	Size	Format	Support
APPLICATION VENDOR	0800h	8 bytes	ASCII	M
APPLICATION NAME	0801h	32 bytes	ASCII	M
APPLICATION VERSION	0802h	8 bytes	ASCII	M
USER MEDIUM TEXT LABEL	0803h	160 bytes	TEXT	O
TEXT LOCALIZATION IDENTIFIER	0805h	1 byte	BINARY	O
BARCODE	0806h	32 bytes	ASCII	O
MEDIA POOL	0808h	160 bytes	TEXT	O
APPLICATION FORMAT VERSION	080Bh	16 bytes	ASCII	M
MEDIUM GLOBALLY UNIQUE IDENTIFIER	0820h	36 bytes	BINARY	O
MEDIA POOL GLOBALLY UNIQUE IDENTIFIER	0821h	36 bytes	BINARY	O

When accessing these attributes, the PARTITION NUMBER field in the READ ATTRIBUTE and WRITE ATTRIBUTE SCSI commands shall be set to 0.

IMPORTANT NOTE: The Mandatory attributes are required to be set by the application which formats the volume. Some storage technology may have insufficient available capacity to store all the attributes in MAM, in which case writing the Mandatory attributes should take precedence over the Optional attributes. However an implementation which attempts to mount the volume should not fail just because these attributes are not set or are unreadable.

10.4.1 Application Vendor

This attribute shall be set to indicate the manufacturer of the LTFS software which formatted the volume. It shall be consistent with the Company name (if any) used in the Creator format in LTFS label and index constructs (see Section 7.2 Creator format). The attribute shall be left-aligned, and shall be padded with ASCII space (20h) characters if the company name is less than 8 characters in length. If the company name exceeds 8 ASCII characters then the 8 left-most characters of the name shall be used.

10.4.2 Application Name

This attribute shall be set to the ASCII string "LTFS", left-aligned and followed by at least one ASCII space (20h) character. This may be followed by a vendor-specific ASCII string further identifying the application, also left-aligned and padded with ASCII space characters. If no further identification is desired then ASCII space characters shall be added to pad to the width of the field. Both of the following are valid uses of this attribute:

"LTFS "
"LTFS Standalone XYZ "

10.4.3 Application Version

This attribute shall be set to indicate the application version used to format the volume and shall be consistent with the Version identifier (if any) used in the Creator format in LTFS label and index constructs (see Section 7.2 Creator format). The attribute shall be left-aligned and padded with ASCII

space (20h) characters. The LTFS format specification does not define any particular style or content for the value of this attribute.

10.4.4 Text Localization Identifier

This defines the character set used for the User Medium Text Label attribute (Section 10.4.5 User Medium Text Label), in accordance with the table in the T10/SPC-4 draft standard (SPC-4 r36e Table 448). If this attribute is not set then the default assumed value shall be ASCII (value 00h).

NOTE: It is strongly recommended that the attribute should be set to indicate UTF-8 encoding (value 81h) for compatibility with the encoding used in the rest of the LTFS format.

10.4.5 User Medium Text Label

This attribute may be used to record the volume name. If set, it shall be left-aligned and null-terminated, and its value should be consistent with the value of the **name** element for the root directory element in an index construct (see Section 9.2 Index). If the number of bytes required to store the root directory name exceeds the available attribute storage size of 160 bytes, then the name stored in the attribute shall be truncated at the most appropriate character boundary. If this attribute is set, and the **name** is updated by writing to the VEA ltfs.volumeName, then this attribute shall be updated to maintain consistency.

10.4.6 Barcode

It is recommended that this attribute should be set to match the physical cartridge label (if any). If set, it shall be left-aligned and padded with ASCII space (20h) characters.

NOTE: This attribute is related to the volume identifier in the VOL1 label (see Section 8.1.1) but without the restriction of six characters; the attribute can hold up to 32 characters.

10.4.7 Media Pool

This attribute may be set to a media pool name and/or additional information as specified in Annex F.4.

10.4.8 Application Format Version

This attribute shall be set to indicate the version of the LTFS format specification with which this volume was formatted. It shall be consistent with the version attribute of the **ltfslabel** element as found in the LTFS label construct (see Section 8.1.2 LTFS Label). It shall be left-aligned and padded with ASCII space (20h) characters.

NOTE: In the special case where a volume is migrated to a newer version of the format, this attribute should be updated to continue to provide an accurate view of the volume. In this case, the attribute may no longer be consistent with the version attribute of the ltfslabel element.

10.4.9 Medium Globally Unique Identifier

This attribute may be used to store the volume UUID, generated when a volume is formatted. It provides access to the UUID of the volume without requiring it to be mounted.

If implemented, the value shall be stored according to the format defined in Section 7.8 UUID format and shall be stored without null termination. The value shall be consistent with the **volumeuuid** value stored in the LTFS volume label (Section 8.1.2 LTFS Label) and in the LTFS index (Section 9.2.3 Required elements for every index).

10.4.10 Media Pool Globally Unique Identifier

This attribute may be set to a media pool UUID as specified in Annex F.4.

10.4.11 Example attributes

An implementation that populates all of the attributes described in Section 10.4 Use of Host-type

Attributes for LTFS would follow the pattern shown in Table 20:

Table 20 — Example of Host-type Attributes

Name	1	2	3	4	5	6	7	8	...	...
Application Vendor	“H”	“P”	20h	20h	20h	20h	20h	20h		
Application Name	“L”	“T”	“F”	“S”	20h	20h	20h	20h	...	20h
Application Version	“1”	“.”	“2”	“.”	“3”	20h	20h	20h		
User Medium Text Label	“M”	“y”	“T”	“a”	“p”	“e”	“V”	“o”	“l”	00h
Text Localization Identifier	81h									
Barcode	“A”	“B”	“1”	“2”	“3”	“4”	“L”	“5”	…	20h
Media Pool	“A”	“n”	“i”	“m”	“a”	“t”	“i”	“o”	“n”	00h
Application Format Version	“2”	“.”	“2”	“.”	“0”	20h	20h	20h	…	20h
Medium Globally Unique Identifier	“3”	“0”	“a”	“9”	“1”	“a”	“0”	“8”	“-“	…
Media Pool Globally Unique Identifier	“8”	“c”	“5”	“5”	“c”	“1”	“4”	“1”	“-”	“...”

10.5 Volume Advisory Locking

The Volume Advisory Locking state of the volume is stored in MAM attribute 1623h, as shown in Table 21 below. This attribute falls within the “Vendor-specific Host-type” range of attributes; the identifier has been chosen to minimize the likelihood of collision with a different application.

Table 21 — Volume Locked MAM Attribute

Attribute Name	Identifier	Size	Format	Support
VOLUME LOCKED	1623h	1 byte	BINARY	M

The MAM attribute shall be set to one of the values shown in Table 22.

Table 22 — Volume Locked MAM Attribute Values

Value	Meaning
0x00	Volume is unlocked (default state)
0x01	Volume is locked at user request
0x02	Volume is locked due to a permanent write error, location is not specified
0x03	Volume is permanently locked at user request
0x04	Volume is locked due to a permanent write error in the data partition
0x05	Volume is locked due to a permanent write error in the index partition
0x06	Volume is locked due to permanent write errors in both partitions

The value 0x02 is retained for backwards compatibility with version 2.3 of the specification but is deprecated for implementations claiming support for version 2.4 or later.

When a volume is formatted, the attribute shall be set to the unlocked state. If a volume does not report this MAM attribute or does not support MAM at all, the locked state can only be determined by reading the index. The **volumelockstate** element which may be stored in the volume index (see Section 9.2.4) shall be treated as definitive except in the case where the MAM attribute indicates that the volume encountered a permanent write error (in which case the index cannot be trusted).

If a volume has been locked due to a permanent write error, an implementation may choose whether to require recovery before mounting or to allow mounting while still in the locked error state. A volume in this state shall be mounted as read-only using the highest generation index available on the tape in either

partition. If the volume is later recovered using some other means, the MAM attribute should be reset to the unlocked state.

The attribute shall be stored for the index partition (i.e. the PARTITION field in the SCSI WRITE ATTRIBUTE command shall correspond to the index partition of the volume).

Annex A (normative) LTFS Label XML Schema

This annex shows the LTFS Label XML Schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="ltfslabel">
      <xs:complexType>
         <xs:all>
            <xs:element name="creator" type="xs:string"/>
            <xs:element name="formattime" type="datetime"/>
            <xs:element name="volumeuuid" type="uuid"/>
            <xs:element name="location">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="partition" type="partitionid"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
            <xs:element name="partitions">
               <xs:complexType>
                  <xs:all>
                     <xs:element name="index" type="partitionid"/>
                     <xs:element name="data" type="partitionid"/>
                  </xs:all>
               </xs:complexType>
            </xs:element>
            <xs:element  name="blocksize"  type="blocksize"/>
            <xs:element name="compression" type="xs:boolean"/>
         </xs:all>
         <xs:attribute name="version" use="required" type="version"/>
      </xs:complexType>
   </xs:element>

   <xs:simpleType name="blocksize">
      <xs:restriction base="xs:integer">
         <xs:minInclusive value="4096"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:simpleType name="version">
      <xs:restriction base="xs:string">
         <xs:pattern value="[0-9]+\.[0-9]+\.[0-9]+"/>
         <xs:enumeration value="2.5.0"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:simpleType  name="datetime">
      <xs:restriction base="xs:string">
         <xs:pattern
            value="[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]{2}\.[0-9]{9}Z"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:simpleType name="partitionid">
      <xs:restriction base="xs:string">
         <xs:pattern value="[a-z]"/>
      </xs:restriction>
   </xs:simpleType>
```

```
   <xs:simpleType name="uuid">
      <xs:restriction base="xs:string">
         <xs:pattern
            value="[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-
9]{12}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

Annex B (normative) LTFS Index XML Schemas

This annex contains the XML schemas for Full and Incremental LTFS Indexes.

B.1 LTFS Full Index XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="ltfsindex">
      <xs:complexType>
         <xs:all>
            <xs:element name="creator" type="xs:string" />
            <xs:element name="comment" type="xs:string" minOccurs="0" />
            <xs:element name="volumeuuid" type="uuid" />
            <xs:element name="generationnumber" type="xs:nonNegativeInteger" />
            <xs:element name="updatetime" type="datetime" />
            <xs:element name="location" type="tapeposition" />
            <xs:element name="previousgenerationlocation" type="tapeposition" minOccurs="0" />
            <xs:element name="previousincrementallocation" type="tapeposition" minOccurs="0" />
            <xs:element name="allowpolicyupdate" type="xs:boolean" />
            <xs:element name="dataplacementpolicy" type="policy" minOccurs="0" />
            <xs:element name="volumelockstate" type="locktype" minOccurs="0" />
            <xs:element name="highestfileuid" type="xs:nonNegativeInteger" />
            <xs:element ref="directory" />
         </xs:all>
         <xs:attribute name="version" use="required" type="version" />
      </xs:complexType>
   </xs:element>
   <xs:element name="directory">
      <xs:complexType>
         <xs:all>
            <xs:element name="fileuid" type="xs:nonNegativeInteger" />
            <xs:element name="name" type="nametype" />
            <xs:element name="creationtime" type="datetime" />
            <xs:element name="changetime" type="datetime" />
            <xs:element name="modifytime" type="datetime" />
            <xs:element name="accesstime" type="datetime" />
            <xs:element name="backuptime" type="datetime" />
            <xs:element name="readonly" type="xs:boolean" />
            <xs:element ref="extendedattributes" minOccurs="0" />
            <xs:element name="contents">
               <xs:complexType>
                  <xs:choice minOccurs="0" maxOccurs="unbounded">
                     <xs:element ref="directory" />
                     <xs:element ref="file" />
                  </xs:choice>
               </xs:complexType>
            </xs:element>
         </xs:all>
      </xs:complexType>
   </xs:element>
   <xs:element name="file">
      <xs:complexType>
         <xs:all>
            <xs:element name="fileuid" type="xs:nonNegativeInteger" />
            <xs:element name="name" type="nametype" />
            <xs:element name="length" type="xs:nonNegativeInteger" />
            <xs:element name="creationtime" type="datetime" />
            <xs:element name="changetime" type="datetime" />
            <xs:element name="modifytime" type="datetime" />
            <xs:element name="accesstime" type="datetime" />
            <xs:element name="backuptime" type="datetime" />
            <xs:element name="readonly" type="xs:boolean" />
            <xs:element ref="extendedattributes" minOccurs="0" />
```

```
         <xs:element name="openforwrite" type="xs:boolean" minOccurs="0" />
         <xs:element ref="extenttype" minOccurs="0" />
      </xs:all>
   </xs:complexType>
</xs:element>
<xs:element name="extendedattributes">
   <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:element name="xattr">
            <xs:complexType>
               <xs:all>
                  <xs:element name="key" type="nametype" />
                  <xs:element name="value">
                     <xs:complexType mixed="true">
                        <xs:attribute name="type" default="text">
                           <xs:simpleType>
                              <xs:restriction base="xs:token">
                                 <xs:enumeration value="base64" />
                                 <xs:enumeration value="text" />
                              </xs:restriction>
                           </xs:simpleType>
                        </xs:attribute>
                     </xs:complexType>
                  </xs:element>
               </xs:all>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="extenttype" abstract="true" />
<xs:element name="extentinfo" substitutionGroup="extenttype">
   <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:element name="extent">
            <xs:complexType>
               <xs:all>
                  <xs:element name="partition" type="partitionid" />
                  <xs:element name="startblock" type="xs:nonNegativeInteger" />
                  <xs:element name="byteoffset" type="xs:nonNegativeInteger" />
                  <xs:element name="bytecount" type="xs:positiveInteger" />
                  <xs:element name="fileoffset" type="xs:nonNegativeInteger" />
               </xs:all>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="symlink" type="nametype" substitutionGroup="extenttype" />
<xs:complexType name="policy">
   <xs:sequence>
      <xs:element name="indexpartitioncriteria">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="size" type="xs:nonNegativeInteger" />
               <xs:element name="name" type="nametype" minOccurs="0" maxOccurs="unbounded" />
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="tapeposition">
   <xs:all>
      <xs:element name="partition" type="partitionid" />
      <xs:element name="startblock" type="xs:nonNegativeInteger" />
   </xs:all>
</xs:complexType>
```

```
   <xs:complexType name="nametype">
      <xs:simpleContent>
         <xs:extension base="xs:string">
            <xs:attribute name="percentencoded" type="xs:boolean" default="false" />
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
   <xs:simpleType name="version">
      <xs:restriction base="xs:string">
         <xs:pattern value="[0-9]+\.[0-9]+\.[0-9]+" />
         <xs:enumeration value="2.5.0" />
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="datetime">
      <xs:restriction base="xs:string">
         <xs:pattern
            value="[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]{2}\.[0-9]{9}Z" />
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="partitionid">
      <xs:restriction base="xs:string">
         <xs:pattern value="[a-z]" />
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="locktype">
      <xs:restriction base="xs:string">
         <xs:enumeration value="unlocked" />
         <xs:enumeration value="locked" />
         <xs:enumeration value="permlocked" />
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="uuid">
      <xs:restriction base="xs:string">
         <xs:pattern
            value="[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

B.2 LTFS Incremental Index XML Schema

The XML Schema Definition (XSD) language has a limited ability to define and validate complex constraint relationships such as those shown below. As a result, the schema definition in this section will perform basic validation of an Incremental Index, but is not capable of detecting certain invalid constructs within the **file** and **directory** elements. Thus an Incremental Index may pass validation by the schema below, but still be invalid according to the definitions in this specification (which always take precedence).

The specific constraints that cannot be expressed in the XSD (but must be adhered to according to the specification) are:

1. A **directory** element in an Increment Index must contain:
   - a **name** element, AND
   - exactly ONE of the following:
     - a **deleted** element
     - a **contents** element
     - a **fileuid** element AND a **contents** element PLUS any other valid **directory** elements (except **deleted**)
2. A **file** element in an Increment Index must contain:
   - a **name** element, AND
   - exactly ONE of the following:
     - a **deleted** element
     - a **fileuid** element PLUS any other valid **file** elements (except **deleted**)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="ltfsincrementalindex">
      <xs:complexType>
         <xs:all>
            <xs:element name="creator" type="xs:string"/>
            <xs:element name="comment" type="xs:string" minOccurs="0"/>
            <xs:element name="volumeuuid" type="uuid"/>
            <xs:element name="generationnumber" type="xs:nonNegativeInteger"/>
            <xs:element name="updatetime" type="datetime"/>
            <xs:element name="location" type="tapeposition"/>
            <xs:element name="previousgenerationlocation" type="tapeposition"/>
            <xs:element name="previousincrementallocation" type="tapeposition" minOccurs="0"/>
            <xs:element name="volumelockstate" type="locktype" minOccurs="0"/>
            <xs:element name="highestfileuid" type="xs:nonNegativeInteger"/>
            <xs:element ref="directory"/>
         </xs:all>
         <xs:attribute name="version" use="required" type="version"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="directory">
      <xs:complexType>
         <xs:all>
            <xs:element name="name" type="nametype"/>
            <xs:element name="fileuid" type="xs:nonNegativeInteger" minOccurs="0" />
            <xs:element name="creationtime" type="datetime" minOccurs="0"/>
            <xs:element name="changetime" type="datetime" minOccurs="0"/>
            <xs:element name="modifytime" type="datetime" minOccurs="0"/>
            <xs:element name="accesstime" type="datetime" minOccurs="0"/>
            <xs:element name="backuptime" type="datetime" minOccurs="0"/>
            <xs:element name="readonly" type="xs:boolean" minOccurs="0"/>
            <xs:element name="deleted" type="xs:string" fixed="" minOccurs="0"/>
            <xs:element ref="extendedattributes" minOccurs="0"/>
            <xs:element ref="contents" minOccurs="0" />
         </xs:all>
      </xs:complexType>
   </xs:element>
   <xs:element name="contents">
      <xs:complexType>
         <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="directory"/>
            <xs:element ref="file"/>
         </xs:choice>
      </xs:complexType>
   </xs:element>
   <xs:element name="file">
      <xs:complexType>
         <xs:all>
            <xs:element name="name" type="nametype"/>
            <xs:element name="fileuid" type="xs:nonNegativeInteger" minOccurs="0" />
            <xs:element name="length" type="xs:nonNegativeInteger" minOccurs="0"/>
            <xs:element name="creationtime" type="datetime" minOccurs="0"/>
            <xs:element name="changetime" type="datetime" minOccurs="0"/>
            <xs:element name="modifytime" type="datetime" minOccurs="0"/>
            <xs:element name="accesstime" type="datetime" minOccurs="0"/>
            <xs:element name="backuptime" type="datetime" minOccurs="0"/>
            <xs:element name="readonly" type="xs:boolean" minOccurs="0"/>
            <xs:element name="deleted" type="xs:string" fixed="" minOccurs="0"/>
            <xs:element ref="extendedattributes" minOccurs="0"/>
            <xs:element name="openforwrite" type="xs:boolean" minOccurs="0"/>
            <xs:element ref="extenttype" minOccurs="0"/>
         </xs:all>
      </xs:complexType>
   </xs:element>
   <xs:element name="extendedattributes">
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
```

```
         <xs:element name="xattr">
            <xs:complexType mixed="true">
               <xs:all>
                  <xs:element name="key" type="nametype"/>
                  <xs:element name="value">
                     <xs:complexType>
                        <xs:attribute name="type" default="text">
                           <xs:simpleType>
                              <xs:restriction base="xs:token">
                                 <xs:enumeration value="base64"/>
                                 <xs:enumeration value="text"/>
                              </xs:restriction>
                           </xs:simpleType>
                        </xs:attribute>
                     </xs:complexType>
                  </xs:element>
               </xs:all>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="extenttype" abstract="true"/>
<xs:element name="extentinfo" substitutionGroup="extenttype">
   <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:element name="extent">
            <xs:complexType>
               <xs:all>
                  <xs:element name="partition" type="partitionid"/>
                  <xs:element name="startblock" type="xs:nonNegativeInteger"/>
                  <xs:element name="byteoffset" type="xs:nonNegativeInteger"/>
                  <xs:element name="bytecount" type="xs:positiveInteger"/>
                  <xs:element name="fileoffset" type="xs:nonNegativeInteger"/>
               </xs:all>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="symlink" type="nametype" substitutionGroup="extenttype"/>
<xs:complexType name="tapeposition">
   <xs:all>
      <xs:element name="partition" type="partitionid"/>
      <xs:element name="startblock" type="xs:nonNegativeInteger"/>
   </xs:all>
</xs:complexType>
<xs:complexType name="nametype">
   <xs:simpleContent>
      <xs:extension base="xs:string">
         <xs:attribute name="percentencoded" type="xs:boolean" default="false"/>
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>
<xs:simpleType name="version">
   <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]+\.[0-9]+\.[0-9]+"/>
      <xs:enumeration value="2.5.0"/>
   </xs:restriction>
</xs:simpleType>
<xs:simpleType name="datetime">
   <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]{2}\.[0-9]{9}Z"/>
   </xs:restriction>
</xs:simpleType>
<xs:simpleType name="partitionid">
   <xs:restriction base="xs:string">
```

```
         <xs:pattern value="[a-z]"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="locktype">
      <xs:restriction base="xs:string">
         <xs:enumeration value="unlocked"/>
         <xs:enumeration value="locked"/>
         <xs:enumeration value="permlocked"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="uuid">
      <xs:restriction base="xs:string">
         <xs:pattern value="[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-
9]{12}"/>
      </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

Annex C (normative) Reserved Extended Attribute definitions

In an LTFS Index, all extended attribute names that start with the prefix “ltfs” with any capitalization are reserved for use by the LTFS Format; i.e., any names starting with a case-insensitive match for the letters “ltfs” are reserved.

Any writer of an LTFS Volume shall only use reserved extended attribute names to store extended attribute values in conformance with the lists in Table C. 1 through Table C. 5. However, two extended attribute namespaces are reserved for implementation-specific information.

The extended attribute namespace “ltfs.permissions.<type>” shall be used only for storing permissions as described in Annex F.2 File Permissions in LTFS.

The extended attribute namespace “ltfs.mediaPool.<attribute>” shall be used only for storing media pool information as described in Annex F.4 LTFS Media Pools.

The extended attribute namespace “ltfs.vendor.X.Y” shall be used for implementation-specific attributes, where X identifies a company, organization or technology standard and Y is an attribute name.

NOTE: The Storage Networking Industry Association (SNIA) maintains a list of registered vendor names at http://www.snia.org/ltfs.

This section describes the meaning of defined, reserved extended attributes.

Support for each of these defined, reserved extended attributes is optional for implementations in compliance with this specification.

C.1 Software Metadata

Table C. 1 describes the extended attribute values for software metadata.

Table C. 1 — Reserved extended attribute definitions: Software metadata

Extended Attribute	Value description
ltfs.softwareProduct	Product name of this software
ltfs.softwareVendor	Software vendor of this software
ltfs.softwareVersion	LTFS version number
ltfs.softwareFormatSpec	LTFS Format spec version supported by this software

C.2 Drive Metadata

Table C. 2 describes the extended attribute values for drive metadata.

Table C. 2 — Reserved extended attribute definitions: Drive metadata

Extended Attribute	Value description
ltfs.driveEncryptionState	Current encryption status of the drive ("true", "false", or "unknown").
ltfs.driveEncryptionMethod	Current encryption method of the drive.
ltfs.driveCaptureDump	Writing any value to this extended attribute shall trigger a drive dump on any implementation that supports this extended attribute.

C.3 Object Metadata

Table C. 3 describes the extended attribute values for object metadata.

Table C. 3 — Reserved extended attribute definitions: Object metadata

Extended Attribute	Value description
ltfs.accessTime	Date and time of last access to object.
ltfs.backupTime	Date and time of last archive or backup of object.
ltfs.changeTime	Date and time of last status change to object.
ltfs.createTime	Date and time of object creation.
ltfs.fileUID	Integer identifier for objects in the filesystem. Guaranteed to be unique within the LTFS Volume.
ltfs.modifyTime	Date and time of last object modification.
ltfs.partition	Partition on which the first extent of the file is stored.
ltfs.startblock	Block address where the first extent of the file is stored.
ltfs.spannedFileOffset	The logical file offset of the first byte of the segment relative to the full file. See Annex F.1 for full description.

C.4 Volume Metadata

Table C. 4 describes the extended attribute values for volume metadata.

Table C. 4 — Reserved extended attribute definitions: Volume metadata

Extended Attribute	Value description
ltfs.commitMessage	On any implementation that supports this extended attribute, writing text to this extended attribute shall trigger a filesystem sync, using the provided text as the comment tag for the index written to the medium. A filesystem sync is an operation that causes all in-memory filesystem changes to be flushed to the storage medium followed by writing of an LTFS index. The sync operation is not required to produce a consistent LTFS Volume, but shall ensure that sufficient data is written to the medium so as to allow the LTFS Volume to be recovered to a consistent state without loss of data. Note that in contrast to ltfs.sync, writing to this extended attribute shall always cause an index to be written to the storage medium, even when there are no changed objects in the filesystem. Reading this extended attribute shall return the commit message in the most recent LTFS index on the storage medium.
ltfs.indexVersion	LTFS format version string for the Index. This string provides a human-readable identifier for the LTFS format version that generated the Index.
ltfs.indexType	Either “Full” or “Incr”, describing the most recent index on the media.
ltfs.indexCreator	Creator string for the Index. This string provides a human- readable identifier for the product that generated the Index. As defined in Section 7.2 Creator format.
ltfs.indexGeneration	Last LTFS Index generation number written to media.
ltfs.indexLocation	Location of the last Index on the media in the form ‘p:l’, where p is an alphabetic character value indicating the internal LTFS partition identifier, and l is the logical block number within the partition. For example, the value ‘a:1000’ indicates that the last Index starts at logical block 1000 on partition a.
ltfs.indexPrevious	Location of the previous Full Index on the media in the form ‘p:l’, where p is an alphabetic character value indicating the internal LTFS partition identifier, and l is the logical block number within the partition. For example, the value ‘b:55’

Extended Attribute	Value description
	indicates that the previous Full Index starts at logical block 55 on partition b.
ltfs.indexPreviousIncremental	Location of the previous Incremental Index on the media since the preceding Full Index, in the form 'p:I' as for ltfs.indexPrevious. Reported as the value 'z:0' if there has not been an Incremental Index since the last Full Index.
ltfs.incrIndexCount	Number of Incremental Indexes on the media since the last Full Index.
ltfs.indexTime	Date and time of when last LTFS Index was written to media.
ltfs.labelVersion	LTFS format version string for the LTFS label. This string provides a human-readable identifier for the LTFS format version that generated the LTFS label.
ltfs.labelCreator	Creator string for the LTFS Label. This string provides a human- readable identifier for the product that generated the LTFS Label. As defined in Section 7.2 Creator format.
ltfs.partitionMap	The on media partition layout for the LTFS Volume. Value is of the form "W:x,Y:z" where W and Y have the value 'I' indicating an index partition, or 'D' indicating a data partition. x and y are an alphabetic character value indicating the internal LTFS partition identifier. For example, the value "I:a,D:b" indicates that LTFS Partition 'a' is used as the index partition, and LTFS Partition 'b' is used as the data partition.
ltfs.policyAllowUpdate	Indicates whether the data placement policy for the volume may be updated.
ltfs.policyExists	Indicates whether a data placement policy has been set for the volume.
ltfs.policyMaxFileSize	Maximum file size for files that match the data placement policy for the volume.
ltfs.sync	On any implementation that supports this extended attribute, writing any value to this extended attribute shall trigger a filesystem sync, except as noted below. A filesystem sync is an operation that causes all in-memory filesystem changes to be flushed to the storage medium followed by writing of an LTFS index. The sync operation is not required to produce a consistent LTFS Volume, but shall ensure that sufficient data is written to the medium so as to allow the LTFS Volume to be recovered to a consistent state without loss of data. An implementation may define its own commit message to be used in the index written by this operation. If there are no changed objects in the filesystem (i.e. the most recent index on the storage medium is consistent with the in-memory index) then an implementation is not required to write another index. Reading this extended attribute shall trigger the same behavior.
ltfs.volumeBlocksize	Blocksize for the LTFS Volume specified at format time.
ltfs.volumeCompression	Compression setting for the LTFS Volume.
ltfs.volumeFormatTime	Date and time when the LTFS Volume was formatted.
ltfs.volumeName	Name of the LTFS Volume.
ltfs.volumeSerial	Serial number for the LTFS Volume specified at format time.
ltfs.volumeUUID	UUID for the LTFS Volume.
ltfs.mamBarcode	The MAM attribute value stored as BARCODE
ltfs.mamApplicationVendor	The MAM attribute value stored as APPLICATION VENDOR
ltfs.mamApplicationVersion	The MAM attribute value stored as APPLICATION VERSION
ltfs.mamApplicationFormat Version	The MAM attribute value stored as APPLICATION FORMAT VERSION
ltfs.volumeLockState	Reflects the protected state of the volume (see Section 9.2.19 Volume Advisory Locking). A value of 0 means the volume may be modified; a non-zero value indicates that the volume has been locked against further modifications.

<table>
<tr><th>Extended Attribute</th><th>Value description</th></tr>
<tr><td></td><td>
This attribute may be written to change the volumelockstate element in the index; the implementation should update the corresponding MAM attribute accordingly (see Section 10.5).

An application may report various forms of protection by encoding them into these bit fields:

<table>
<tr><th>Bit</th><th>31 ... 8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Field</td><td>0
(Reserved)</td><td>IPPWE</td><td>DPPWE</td><td>PWE</td><td>PERSWP</td><td>PRMWP</td><td>Physical Write Protect</td><td>Perm-Locked</td><td>Locked</td></tr>
</table>

The Locked and PermLocked bits correspond to the index information described in Section 9.2.4 for the volumelockstate element and are mutually exclusive.

The Physical Write Protect bit should be reported as 1 if writing to the volume has been prevented by some physical means (for example sliding a Protect tab on a tape cartridge). It cannot be changed by writing this attribute and should be ignored in that case.

The Persistent Write Protect (PERSWP) and Permanent Write Protect (PRMWP) bits should be reported as 1 if the tape drive has been set into the corresponding mode. Note that support for PERSWP and PRMWP is dependent on the underlying tape drive technology. An application may support changing PERSWP and PRMWP, contingent on the underlying device also supporting those fields. Refer to the T10 draft specification SSC4 Section 4.2.16 “Write Protection” for further details.

The Index Partition Permanent Write Error (IPPWE) and Data Partition Permanent Write Error (DPPWE) bits should be reported as 1 if the tape drive encountered a permanent write error whilst writing in the corresponding partition.

The non-partition-specific Permenent Write Error (PWE) bit is preserved for backwards compatibility with version 2.3 but is deprecated for implementations claiming compliance with version 2.4 of the specification. IPPWE and/or DPPWE should be used instead (based on the value stored in the MAM attribute 1623h, see Section 10.5 Volume Advisory Locking).

The IPPWE, DPPWE and PWE bits cannot be changed by writing this attribute, and should be ignored in that case. If one or more of these bits are reported as one then the Locked bit should also be reported as 1.
</td></tr>
</table>

NOTE 1: The USER MEDIUM TEXT LABEL MAM attribute is available as ltfs.volumeName.

NOTE 2: The VEAs ltfs.softwareVendor, ltfs.softwareProduct, ltfs.softwareVersion, and ltfs.softwareFormatSpec refer to the currently executing software, whereas the above names ltfs.mamApplicationVendor etc refers to the values stored in the MAM at format time.

NOTE 3: Setting or updating the VEA ltfs.mamBarcode after the volume has been formatted should update the MAM attribute but shall not modify the VOL1 label nor the value reported for the VEA ltfs.volumeSerial.

C.5 Media Metadata

Table C. 5 describes the extended attribute values for media metadata.

Table C. 5 — Reserved extended attribute definitions: Media metadata

Extended Attribute	Value description
ltfs.mediaBeginningMediumPasses	Total number of times the beginning of medium position has been passed. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaDataPartitionAvailableSpace	Total available space in the Data Partition on the medium. Value is an integer count measured in units of 1048576 bytes.
ltfs.mediaDataPartitionTotalCapacity	Total capacity of the Data Partition on the medium. Value is an integer count measured in units of 1048576 bytes.
ltfs.mediaDatasetsRead	Total number of datasets read from the medium over the lifetime of the media. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaDatasetsWritten	Total number of datasets written to the medium over the lifetime of the media. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaEfficiency	An overall measure of the condition of the loaded media. The value 0x00 indicates that the condition is unknown. The range of known values is from 0x01 (best condition) to 0xFF (worst condition). If the storage hardware cannot report this data the value will be −1.
ltfs.mediaIndexPartitionAvailableSpace	Total available space in the Index Partition on the medium. Value is an integer count measured in units of 1048576 bytes.
ltfs.mediaIndexPartitionTotalCapacity	Total capacity of the Index Partition on the medium. Value is an integer count measured in units of 1048576 bytes.
ltfs.mediaLoads	Number of times the media has been loaded in a drive. For example, with tape media this will be the tread count. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaMBRead	Total number of megabytes of logical object data read from the medium after compression over the lifetime of the media. The value shall be rounded up to the next whole megabyte. The value reported shall include bytes read as part of reading filemarks from the media. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaMBWritten	Total number of megabytes of logical object data written to the medium after compression over the lifetime of the media. The value shall be rounded up to the next whole megabyte. The value reported shall include bytes written as part of writing filemarks to the media. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaMiddleMediumPasses	Total number of times the physical middle position of the user data region of medium has been passed. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaEncrypted	True if the Medium is encrypted or False if not.
ltfs.mediaPermanentReadErrors	Total number of unrecovered data read errors over the lifetime of the media. This is the total number of times that a read type command terminated with a sense key of MEDIUM ERROR, HARDWARE ERROR, or equivalent over the media life. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaPermanentWriteErrors	Total number of unrecovered data write errors over the lifetime of the media. This is the total number of times that

Extended Attribute	Value description
	a write type command terminated with a sense key of MEDIUM ERROR, HARDWARE ERROR, or equivalent over the media life. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaPreviousPermanentReadErrors	Total number of unrecovered read errors that occurred during the previous load of the media. This is the total number of times that a read type command terminated with a sense key of MEDIUM ERROR, HARDWARE ERROR, or equivalent during the previous load session. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaPreviousPermanentWriteErrors	Total number of unrecovered write errors that occurred during the previous load of the media. This is the total number of times that a write type command terminated with a sense key of MEDIUM ERROR, HARDWARE ERROR, or equivalent during the previous load session. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaRecoveredReadErrors	Total number of recovered read errors for the lifetime of the media. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaRecoveredWriteErrors	Total number of recovered data write correction errors over the lifetime of the media. If the storage hardware cannot report this data the value will be −1.
ltfs.mediaStorageAlert	A 64bit value containing alert flags for the storage system. For data tape media this value is equal to the standard tape alert flags. The standard tape alert flags are cleared when read, but this flag's values are latched after a flag is raised once. When a 64bit value is written to this EA the flags that correspond to the 1 bits written are cleared; this is the only way that flags are cleared. If the storage hardware cannot report this data the value will be the string "UNKNOWN".

Annex D (informative) Example of Valid Simple Complete LTFS Volume

Figure D.1 shows the content of a simple LTFS volume. This volume contains three files “A”, “B”, and “C”. File “A” is comprised of three extents. Files “B” and “C” each have one extent.

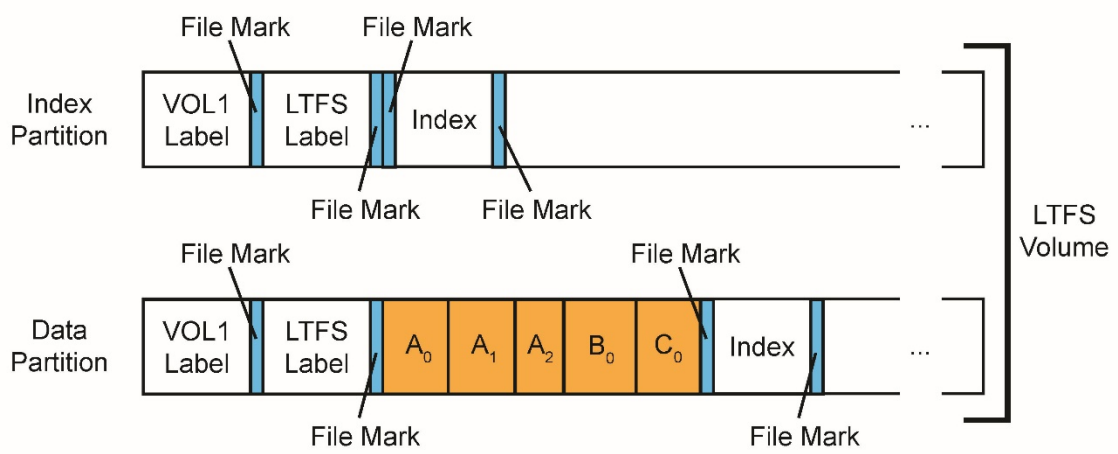


Figure D.1 — Content of a simple LTFS volume

Annex E (informative) Complete Example LTFS Full Index

This annex shows a complete example of an LTFS Full Index that includes the important features of the Index format.

In this index:

- The file directory2/binary_file.bin has a length (20000000 bytes) greater than that of its extent list (10485760 bytes). The extra length is implicitly filled with zero bytes as described in Section 6.1 Extent Lists.
- Block 8 of partition 'b' is shared. The first 720000 bytes of the block are used by directory2/binary_file.bin and directory2/binary_file2.bin. The next 105008 bytes are used only by directory2/binary_file2.bin. This form of sharing data between files is described in Section 6.3.4 Shared Data.
- The file partialfile.bin was still open for writing at the time that the index was written. The length and extent information reflect the file properties at that point in time. A cleanly unmounted volume would not have any open files so the openforwrite element (see Section 9.2.9) would not normally be present.

```
<?xml version="1.0" encoding="UTF-8"?>
<ltfsindex version="2.5.0">
   <creator>IBM LTFS 2.5.0 - Linux - ltfs</creator>
   <volumeuuid>5d217f76-53e6-4d6f-91d1-c4213d94a742</volumeuuid>
   <generationnumber>3</generationnumber>
   <updatetime>2018-10-01T11:45:27.150534438Z</updatetime>
   <location>
      <partition>a</partition>
      <startblock>6</startblock>
   </location>
   <previousgenerationlocation>
      <partition>b</partition>
      <startblock>20</startblock>
   </previousgenerationlocation>
   <allowpolicyupdate>true</allowpolicyupdate>
   <dataplacementpolicy>
      <indexpartitioncriteria>
         <size>1048576</size>
         <name>*.txt</name>
      </indexpartitioncriteria>
   </dataplacementpolicy>
   <highestfileuid>11</highestfileuid>
   <directory>
      <fileuid>1</fileuid>
      <name>LTFS Volume Name</name>
      <readonly>false</readonly>
      <creationtime>2013-02-16T19:13:42.986549106Z</creationtime>
      <changetime>2013-02-16T19:13:47.517309274Z</changetime>
      <modifytime>2013-02-16T19:13:47.517309274Z</modifytime>
      <accesstime>2013-02-16T19:13:42.986549106Z</accesstime>
      <backuptime>2013-02-16T19:13:42.986549106Z</backuptime>
      <contents>
         <directory>
            <fileuid>2</fileuid>
            <name>directory1</name>
            <readonly>false</readonly>
            <creationtime>2013-02-16T19:13:43.006599071Z</creationtime>
            <changetime>2013-02-16T19:13:48.524075283Z</changetime>
```

```
      <modifytime>2013-02-16T19:13:46.514736591Z</modifytime>
      <accesstime>2013-02-16T19:13:43.006599071Z</accesstime>
      <backuptime>2013-02-16T19:13:43.006599071Z</backuptime>
      <extendedattributes>
         <xattr>
            <key>binary_xattr</key>
            <value  type="base64">yDaaBPBdIUqMhg==</value>
         </xattr>
         <xattr>
            <key>empty_xattr</key>
            <value/>
         </xattr>
      </extendedattributes>
      <contents>
         <directory>
            <fileuid>3</fileuid>
            <name>subdir1</name>
            <readonly>false</readonly>
            <creationtime>2013-02-16T19:13:46.514736591Z</creationtime>
            <changetime>2013-02-16T19:13:46.514736591Z</changetime>
            <modifytime>2013-02-16T19:13:46.514736591Z</modifytime>
            <accesstime>2013-02-16T19:13:46.514736591Z</accesstime>
            <backuptime>2013-02-16T19:13:46.514736591Z</backuptime>
            <contents/>
         </directory>
      </contents>
   </directory>
   <directory>
      <fileuid>4</fileuid>
      <name>directory2</name>
      <readonly>false</readonly>
      <creationtime>2013-02-16T19:13:43.007872849Z</creationtime>
      <changetime>2013-02-16T19:13:46.512350773Z</changetime>
      <modifytime>2013-02-16T19:13:46.512350773Z</modifytime>
      <accesstime>2013-02-16T19:13:43.007872849Z</accesstime>
      <backuptime>2013-02-16T19:13:43.007872849Z</backuptime>
      <contents>
         <file>
            <fileuid>5</fileuid>
            <name>sparse_file.bin</name>
            <length>20000000</length>
            <readonly>false</readonly>
            <creationtime>2013-02-16T19:13:45.012828533Z</creationtime>
            <changetime>2013-02-16T19:13:46.509553802Z</changetime>
            <modifytime>2013-02-16T19:13:46.509553802Z</modifytime>
            <accesstime>2013-02-16T19:13:45.012828533Z</accesstime>
            <backuptime>2013-02-17T19:15:34.032137221Z</backuptime>
            <extentinfo>
               <extent>
                  <partition>b</partition>
                  <startblock>8</startblock>
                  <byteoffset>0</byteoffset>
                  <bytecount>720000</bytecount>
                  <fileoffset>0</fileoffset>
               </extent>
               <extent>
                  <partition>b</partition>
                  <startblock>18</startblock>
                  <byteoffset>0</byteoffset>
                  <bytecount>600000</bytecount>
```

```
            <fileoffset>720000</fileoffset>
          </extent>
          <extent>
            <partition>b</partition>
            <startblock>9</startblock>
            <byteoffset>271424</byteoffset>
            <bytecount>9165760</bytecount>
            <fileoffset>1375000</fileoffset>
          </extent>
        </extentinfo>
      </file>
      <file>
        <fileuid>6</fileuid>
        <name>binary_file2.bin</name>
        <length>825008</length>
        <readonly>false</readonly>
        <creationtime>2013-02-16T19:13:46.512350773Z</creationtime>
        <changetime>2013-02-16T19:13:46.513510263Z</changetime>
        <modifytime>2013-02-16T19:13:46.513510263Z</modifytime>
        <accesstime>2013-02-16T19:13:46.000000000Z</accesstime>
        <backuptime>2013-02-16T19:13:46.512350773Z</backuptime>
        <extentinfo>
          <extent>
            <partition>b</partition>
            <startblock>8</startblock>
            <byteoffset>0</byteoffset>
            <bytecount>825008</bytecount>
            <fileoffset>0</fileoffset>
          </extent>
        </extentinfo>
      </file>
    </contents>
  </directory>
  <file>
    <fileuid>7</fileuid>
    <name>testfile.txt</name>
    <length>5</length>
    <readonly>false</readonly>
    <creationtime>2013-02-16T19:13:44.009581288Z</creationtime>
    <changetime>2013-02-16T19:13:49.532111261Z</changetime>
    <modifytime>2013-02-16T19:13:49.532111261Z</modifytime>
    <accesstime>2013-02-16T19:13:49.527726902Z</accesstime>
    <backuptime>2013-02-16T19:13:44.009581288Z</backuptime>
    <extendedattributes>
      <xattr>
        <key>author_name</key>
        <value>Michael Richmond</value>
      </xattr>
    </extendedattributes>
    <extentinfo>
      <extent>
        <partition>a</partition>
        <startblock>4</startblock>
        <byteoffset>0</byteoffset>
        <bytecount>5</bytecount>
        <fileoffset>0</fileoffset>
      </extent>
    </extentinfo>
  </file>
  <file>
```

```
   <fileuid>8</fileuid>
   <name>read_only_file</name>
   <length>0</length>
   <readonly>true</readonly>
   <creationtime>2013-02-16T19:13:47.517309274Z</creationtime>
   <changetime>2013-02-16T19:13:47.519534438Z</changetime>
   <modifytime>2013-02-16T19:13:47.000000000Z</modifytime>
   <accesstime>2013-02-16T19:13:47.000000000Z</accesstime>
   <backuptime>2013-02-16T19:13:47.517309274Z</backuptime>
   <extendedattributes>
      <xattr>
         <key>author_name</key>
         <value>Brian Biskeborn</value>
      </xattr>
   </extendedattributes>
</file>
<file>
   <fileuid>9</fileuid>
   <name>symlink_file</name>
   <length>27</length>
   <readonly>false</readonly>
   <creationtime>2013-02-16T19:49:11.247309274Z</creationtime>
   <changetime>2013-02-16T19:49:11.249534438Z</changetime>
   <modifytime>2013-02-16T19:49:11.000000000Z</modifytime>
   <accesstime>2013-02-16T19:49:11.000000000Z</accesstime>
   <backuptime>2013-02-16T19:49:11.247309274Z</backuptime>
   <extendedattributes>
      <xattr>
         <key>author_name</key>
         <value>David Pease</value>
      </xattr>
   </extendedattributes>
   <symlink>directory2/binary_file2.bin</symlink>
</file>
<file>
   <fileuid>10</fileuid>
   <name percentencoded="true">Testfile%3A1.txt</name>
   <length>13652</length>
   <readonly>false</readonly>
   <creationtime>2014-07-02T09:13:27.730249274Z</creationtime>
   <changetime>2014-07-02T09:13:27.749534438Z</changetime>
   <modifytime>2014-07-02T09:13:27.000000000Z</modifytime>
   <accesstime>2014-07-02T09:13:27.000000000Z</accesstime>
   <backuptime>2014-07-02T09:13:27.730249274Z</backuptime>
   <extendedattributes>
      <xattr>
         <key>author_name</key>
         <value>Chris Martin</value>
      </xattr>
      <xattr>
         <key percentencoded="true">Sample%3Aencoded_name</key>
         <value>Value: is never %-encoded!</value>
      </xattr>
   </extendedattributes>
   <extentinfo>
      <extent>
         <partition>b</partition>
         <startblock>20</startblock>
         <byteoffset>0</byteoffset>
         <bytecount>13652</bytecount>
```

```
            <fileoffset>0</fileoffset>
          </extent>
        </extentinfo>
      </file>
      <file>
        <fileuid>11</fileuid>
        <name>partialfile.bin</name>
        <length>10485760</length>
        <readonly>false</readonly>
        <openforwrite>true</openforwrite>
        <creationtime>2016-02-05T11:45:27.123449274Z</creationtime>
        <changetime>2016-02-05T11:45:27.150534438Z</changetime>
        <modifytime>2016-02-05T11:45:27.150534438Z</modifytime>
        <accesstime>2016-02-05T11:45:27.150534438Z</accesstime>
        <backuptime>2016-02-05T11:45:27.150534438Z</backuptime>
        <extentinfo>
          <extent>
            <partition>b</partition>
            <startblock>21</startblock>
            <byteoffset>0</byteoffset>
            <bytecount>10485760</bytecount>
            <fileoffset>0</fileoffset>
          </extent>
        </extentinfo>
      </file>
    </contents>
  </directory>
</ltfsindex>
```

Annex F (normative) Interoperability Recommendations

This annex describes recommended practices to enable LTFS interoperability.

F.1 Spanning Files across Multiple Tape Volumes in LTFS

LTFS is designed so that tape volumes are self-contained and self-describing. However, there may be times when a file simply does not fit on a single tape and it is necessary to span the file across two or more tapes. A standard method of identifying such files and indicating where the other segments of them may be found is important to guarantee interoperability between LTFS implementations.

The term spanned file refers to a file which is represented by multiple files segments, each typically on a separate tape volume. The term file segment refers to a file on a single LTFS volume which is actually a part of a spanned file.

F.1.1 File Naming

A file segment shall have a name that consists of the true (base) file name with a suffix that identifies the file as a segment of a spanned file. The format of the suffix shall be:

-LTFSseg*n* where capitalization is as shown and *n* is the segment number. Segment numbering shall start at 1 for the first segment of the file (i.e., file offset 0), and increment by 1 for each subsequent segment (in increasing file offset order).

For example, if the true name of a file is foo.txt, and the file is spanned into two segments, the first segment name shall be foo.txt-LTFSseg1 and the second shall be foo.txt-LTFSseg2. The original file name can always be found by stripping the suffix from the stored name.

F.1.2 File Location

Each segment of a spanned file shall have the same directory path on its respective LTFS volume. For example, if the fully qualified LTFS path name to foo.txt-LTFSseg1 is /user/data/foo.txt-LTFSseg1, then the fully qualified path name to foo.txt-LTFSseg2 shall be /user/data/foo.txt-LTFSseg2.

F.1.3 Segment References

Each segment shall be accompanied on its tape volume by a reference to the prior segment and a reference to the next segment, as appropriate. The implementation of these references shall be a symbolic link containing the fully qualified path to the referenced segment. For example, if /user/data/foo.txt-LTFSseg1 is on tape volume 001005 and /user/data/foo.txt-LTFSseg2 is on tape volume 001006, there shall be a symbolic link on tape 001005 named /user/data/foo.txt-LTFSseg2, whose target value is /001006/user/data/foo.txt-LTFSseg2. Similarly, there shall be a symbolic link on tape 001006 named /user/data/foo.txt-LTFSseg1, whose target value is /001005/user/data/foo.txt-LTFSseg1. For files that span more than two tapes (so that multiple preceding and following segments exist for some tapes), the implementation may optionally include segment references to all other segments, but shall minimally include prior and next references (as appropriate).

F.1.4 Extended Attributes

To allow easy identification of spanned file segments, to enable efficient access to those segments, and to verify the correctness of the segmentation, each spanned file segment shall be accompanied by an extended attribute. The name of this extended attribute shall be ltfs.spannedFileOffset, and its value shall be the logical file offset of the first byte of the segment relative to the full file. A file that is not a segment of a spanned file shall not have an ltfs.spannedFileOffset attribute.

Representation of Spanned File Segment Offset Extended Attribute:

```
<xattr>
        <key>ltfs.spannedFileOffset</key>
        <value>offset</value>
</xattr>
```

Where;

offset = the logical file offset of the first byte of the file segment (decimal representation)

Example:

```
<xattr>
        <key>ltfs.spannedFileOffset</key>
        <value>800</value>
</xattr>
```

In this example, the length of file foo.txt is 1000 bytes, and the length of segment foo.txt-LTFSseg1 is 800 bytes. The value of the ltfs.spannedFileOffset attribute for file foo.txt-LTFSseg1 shall be 0 and the value of the ltfs.spannedFileOffset attribute for file foo.txt-LTFSseg2 shall be 800 as shown above.

F.1.5 File Operations

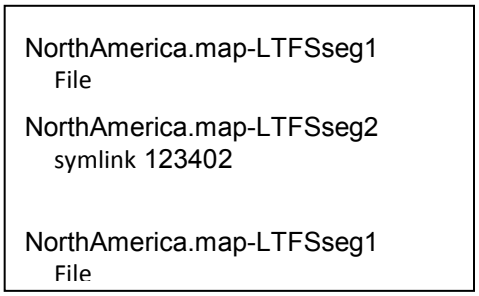
LTFS file system implementations that support creating or modifying spanned files are not required to support the renaming of spanned files nor the random overwrite of bytes within a spanned file. They should support appending to the end of a spanned file and truncation of spanned files (which may be immediately followed by an append), as well as any other file system operations supported for non-spanned files. For every segment of the file, the attributes of the segment represent the state of the file at the time that the segment was written.

F.1.6 Examples

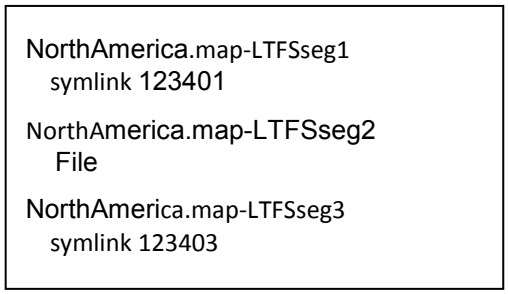
F.1.6.1 Example 1

File “NorthAmerica.map” is too large for 1 tape, so it is to be split across four tapes (tape volume ids 123401 – 123404). The resulting file segments are “NorthAmerica.map-LTFSseg1” through “NorthAmerican.map-LTFSseg4”. It is desired to utilize a minimum number of required symlinks on each tape.

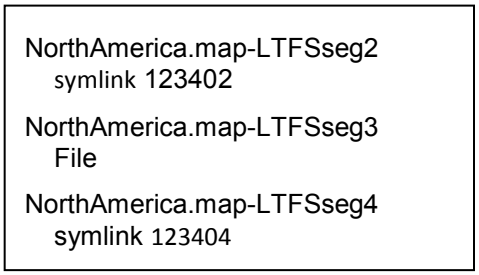
Tape Volume 123401



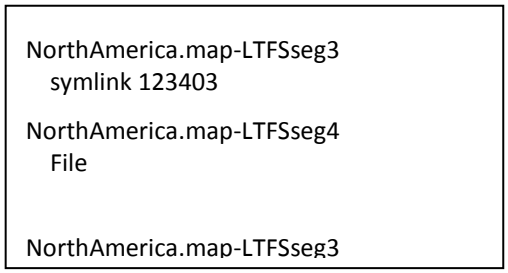
Tape Volume 123402



Tape Volume 123403



Tape Volume 123404



F.1.6.2 Example 2

File “NorthAmerica.map” is too large for 1 tape, so it is to be split across four tapes (tape volume ids 123401 – 123404). The resulting file segments are “NorthAmerica.map-LTFSseg1” through “NorthAmerican.map-LTFSseg4”. It is desired to utilize all of the allowed symlinks on each tape.

Tape Volume 123401

NorthAmerica.map-LTFSseg1
file

NorthAmerica.map-LTFSseg2
symlink 123402

NorthAmerica.map-LTFSseg3
symlink 123403

NorthAmerica.map-LTFSseg4
symlink 123404

Tape Volume 123402

NorthAmerica.map-LTFSseg1
symlink 123401

NorthAmerica.map-LTFSseg2
file

NorthAmerica.map-LTFSseg3
symlink 123403

NorthAmerica.map-LTFSseg4
symlink 123404

Tape Volume 123403

NorthAmerica.map-LTFSseg1
symlink 123401

NorthAmerica.map-LTFSseg2
symlink 123402

NorthAmerica.map-LTFSseg3
file

NorthAmerica.map-LTFSseg4
symlink 123404

Tape Volume 123404

NorthAmerica.map-LTFSseg1
symlink 123401

NorthAmerica.map-LTFSseg2
symlink 123402

NorthAmerica.map-LTFSseg3
symlink 123403

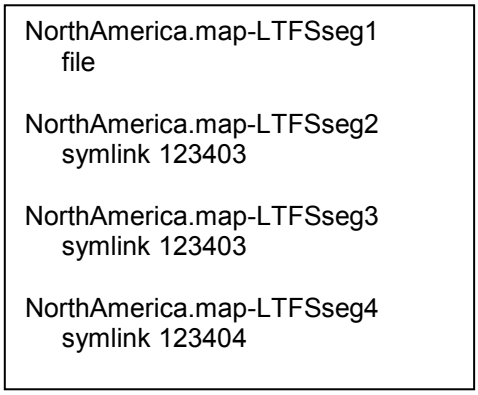
NorthAmerica.map-LTFSseg4
file

F.1.6.3 Example 3

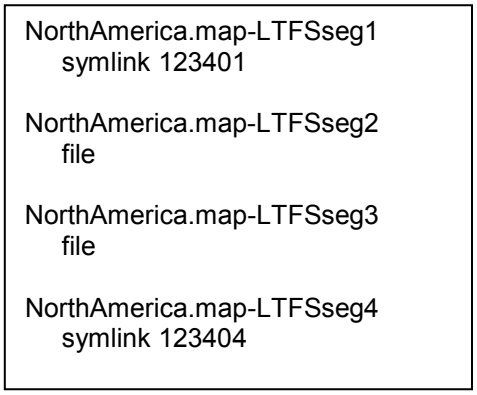
Typically, it is expected that spanned file segments for the same file will reside on different volumes. However, files on a volume may be copied to another volume. One reason for doing so might be to reclaim the space taken up by deleted or overwritten files as well as old indexes. This operation is sometimes referred to as “reclamation”. If the destination of the reclamation operation is a non-empty volume, it is possible that the operation may result in two or more spanned file segments for the same file on the destination volume. In this case, there is potential for a collision between a symlink and the spanned file segment that it points to, because they have the same name and reside at the same location in the directory tree. In this case, the symlink shall not be written.

Given the volumes of Example 2, file segment “NorthAmerica.map-LTFSseg2” has been moved from tape volume 123402 to tape volume 123403. Tape volume 123402 is no longer part of the set of spanned volumes. The symlink on volume 123403 which referenced file segment “NorthAmerica.map-LTFSseg2” has been replaced by the file segment itself. The symlinks on tape volume 123401 and tape volume 123404 have been updated to reference the file segment at the new location.

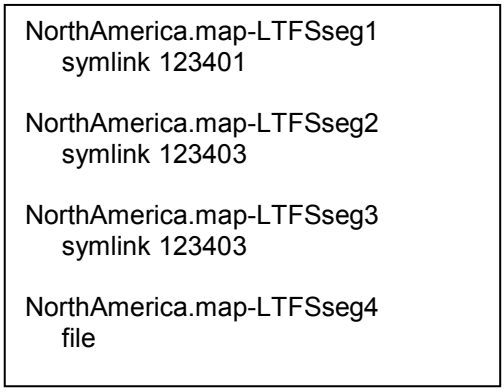
Tape Volume 123401



Tape Volume 123403



Tape Volume 123404



F.2 File Permissions in LTFS

File permissions in interchange media are problematic for two reasons: First, they rely on some type of user (and often group) identification that is usually not transportable except within a narrow scope (typically the same OS in the same location). Second, they are not standard across different types of operating systems, and it is difficult or impossible to accurately convert one type of permission to another (for example, POSIX ACLs to NTFS ACLs or Unix permission bits).

However, there are some instances where storing and enforcing permissions in LTFS may be desirable, including in a single data center where the operating system environment is constant and a central user identification system is used, or when the data on tape is being stored for retention purposes and the permission information is an important attribute of the data to be able to retrieve.

The recommendation is to optionally store permissions in a well-defined, operating system-specific reserved LTFS extended attribute, and optionally honor those permissions if they exist for the operating system environment being used.

The LTFS extended attribute (EA) for File and Directory Permissions is of the form:

ltfs.permissions.<permissiontype>

For example, ltfs.permissions.unix for unix permission bits or ltfs.permissions.ntfsacl for Windows NTFS ACLs. The list of currently identified permission types includes:

unix	Unix permission bits
posixacl	POSIX ACL
ntfsacl	NTFS ACL
nfsv4acl	NFS V4 ACL

Multiple Permission EAs are not precluded for a given file or directory although there can be at most one EA of a given permission type.

It is recommended that an LTFS implementation provide mount-time options that specify whether permissions are stored and/or enforced. When storing of file permissions is enabled, the system-specific permission EA associated with the file should be created or updated (for example, during file creation, chmod processing, icacls command in Windows, etc.). Additionally, when permissions are enforced and a file is accessed, if an appropriate system-specific permission EA exists for the file, it should be used to determine whether the file access is allowed.

F.2.1 Unix Permissions:

Representation of Unix Permissions Extended Attribute:

```
<xattr>
        <key>ltfs.permissions.unix</key>
        <value>uuuuu;ggggg;pppp</value>
</xattr>
```

Where;

```
uuuuu = numeric uid associated with file (decimal representation)
ggggg = numeric gid associated with file (decimal representation)
pppp  = octal representation of Unix permission bits
```

Example:

```
<xattr>
        <key>ltfs.permissions.unix</key>
        <value>12156;1001;0640</value>
```

```
</xattr>
```

This example has read and write permission for the owner, read permission for the group, and no access for others. The owner's user id (uid) is 12156 (pease), and the owner's group id (gid) is 1001 (users).

F.2.2 POSIX ACLs:

Representation of POSIX ACLs Extended Attribute:

```
<xattr>
        <key>ltfs.permissions.posixacl</key>
        <value>uuuu;gggg;[fff];ACE[;...]</value>
</xattr>
```

Where;

uuuuu = numeric uid associated with file ownership (decimal representation)

ggggg = numeric gid associated with file group (decimal representation)

fff = optional file flags which represent the high-order octal digit of the Unix permissions (e.g., a file with the “setuid” bit on would have a flag value of “s--”)

ACE = an Access Control Entry, of the form “[default:]type:[id]:value”; multiple ACEs can exist in an ACL, separated by semicolons (“;”); fields within the ACE are separated by colons (“:”)

default is an optional literal string value “default”

type is string value of either “group”, “user”, “mask”, or “other”

id is an optional decimal representation

value is a 3 character string representation of the permission flags (“r”,”w”,”x” or “-“)

Example:

```
<xattr>
        <key>ltfs.permissions.posixacl</key>
        <value>12156;1001;;user::rw-;group::r--;other::---</value>
</xattr>
```

In this example, the user id 12156 has read/write access. Members of group 1001 have read only access. All other users and group members have no access. There are no special flags.

The next example shows the use of multiple user and group ids as well as special file flags in an ACL:

```
<xattr>
        <key>ltfs.permissions.posixacl</key>
        <value>12156;1001;--t;user::rw-;user:9532:r--;user:1476:rw-;group::r--
;group:17:rw-;mask::rw-;other::---</value>

</xattr>
```

In this example, the user id 12156 has read/write access, the user id 9532 has read access and the user id 1476 has read/write access. Members of group 1001 have read only access and the members of group 17 have read/write access. All other users and group members have no access. The file has the Unix “sticky bit” set (file flag “--t”).

F.2.3 NFSv4 ACLs:

Representation of NFSv4 ACLs Extended Attribute (see *IETF RFC 3530 NFS version 4* protocol) is as

follows:

```
<xattr>
        <key> ltfs.permissions.nfsv4acl</key>
        <value>ACE[;ACE...]</value>
</xattr>
```

Where;

ACE = An Access Control Entry of the form “type:flag:mask:who”; multiple ACEs can exist in an ACL, separated by semicolons (“;”); fields within the ACE are separated by colons (“:”)

type is defined as a unsigned int (uint32_t) (hexadecimal)

flag is defined as a unsigned int (uint32_t) (hexadecimal)

mask is defined as a unsigned int (uint32_t) (hexadecimal)

who is defined as a opaque utf8 mixed case string (utf8str_mixed)

Example:

```
<xattr>
        <key> ltfs.permissions.nfsv4acl</key>
        <value>0x00:0x00:0x02:OWNER@</value>
</xattr>
```

Where the type is ACCESS ALLOWED, the flag is undefined, the mask is WRITE DATA, and the who is “OWNER@”

In this example, the file can only be written by the file owner.

NOTE There can be multiple of the above entries for a given file. For example;

```
<xattr>
        <key> ltfs.permissions.nfsv4acl</key>
        <value>0x00:0x00:0x010002:OWNER@;0x01:0x00:0x01:EVERYONE@</value>
</xattr>
```

Where the type is ACCESS ALLOWED, the flag is undefined, the mask is WRITE DATA and DELETE, and the who is “OWNER@” for the first ACL and the type is ACCESS ALLOWED, the mask is READ DATA and the who is “EVERYONE@” for the 2nd ACL.

In this example, the file can be written and deleted by the file owner and read by everyone.

F.2.4 NTFS ACLs:

Representation of NTFS ACLs Extended Attribute is as follows:

```
<xattr>
        <key>ltfs.permissions.ntfsacl</key>
        <value>osid;gsid;type:flag:mask:sid[;type:flag:mask:sid][…]</value>
</xattr>
```

Where;

osid is the owner sid (UTF8 string)

gsid is the group sid (UTF8 string)

type is defined as a numeric value (hexadecimal)

flag is defined as a numeric value (hexadecimal)

mask is defined as a 32 bit value (hexadecimal)

sid is the Security ID (UTF8 string)

Example:

```
<xattr>
        <key>ltfs.permissions.ntfsacl</key>
        <value>S-1-5-32-545;S-1-5-64-10;0x00:0x00:0x1F0000:S-1-5-32-544</value>
</xattr>
```

Where the owner sid is “S-1-5-32-545”, the group sid is “S-1-5-64-10”, the type is ACCESS ALLOWED, the flag value is zero, the mask is STANDARD_WRITES_ALL and the SID is “S-1-5-32-544”

F.3 Storing File Hash Values in LTFS

It is desireable to have a method of verifying the contents of files stored on an LTFS volume. This is typically done by calculating a checksum or secure hash value for the file. This hash value can be stored in the index, then at a later date the same calculation can be repeated and compared with the value from the index which was stored at the time the file was written.

The mechanism used to generate the hash values, details of how they are maintained and updated, and the process for checking them, are all outside the scope of this format specification. There is no requirement for an implementation to generate or to check hash values. However if an implementer decides to make use of this technique, the extended attributes defined in this section should be used to enable interoperability of implementations.

F.3.1 Extended Attributes

The hash values may optionally be stored using LTFS extended attribute (EA) names of the form

ltfs.hash.<*hashtype*>

where *hashtype* identifies the algorithm used. Table F.1 — Hash Types lists the *hashtype* values are covered by this format specification; all other hashtype values are reserved.

Table F.1 — Hash Types

hashtype	Algorithm reference	Message Digest size (bits)	UTF-8 representation (hex characters)
crc32sum	ANSI X3.66	32	8
md5sum	IETF RFC1321	128	32
sha1sum	FIPS PUB 180-4	160	40
sha256sum	FIPS PUB 180-4	256	64
sha512sum	FIPS PUB 180-4	512	128

In general the algorithms using more bits provide a greater level of confidence (less risk of false matches or “collisions”) but at the expense of increased computation time.

A single file may have multiple EA’s storing different *hashtypes*, but shall have at most one EA for any given *hashtype*.

The *hashtype* EA is undefined (i.e. is not valid) for directory elements and for file elements which contain a symlink element.

F.3.2 Representation

The hash values are stored in the index as xattr key/value pairs. The name of the key shall be the full EA name, and the value shall be a UTF-8 encoded hexadecimal string representation of the message digest, of the length shown in Table F.1 — Hash Types above.

Example:

```
<xattr>
      <key>ltfs.hash.md5sum</key>
      <value>43aba6a17650519558fede41dd10d400</value>
</xattr>
```

F.4 LTFS Media Pools

A storage system may combine LTFS volumes into media pools. One reason for doing so might be to create a file system which spans multiple volumes and whose capacity may be increased by adding volumes to the media pool. If an LTFS volume belongs to a media pool, and that membership is to be persistent outside of the storage system, then there needs to be a way of identifying the media pool to which the LTFS volume belongs.

When working with media pools, because the media pool name may be changed, the media pool UUID is considered to be definitive when determining whether volumes reside in the same media pool.

F.4.1 Media Pool Membership of a Volume

The media pool name to which an LTFS volume belongs may be stored in an extended attribute for the root directory element in an index (see 9.2). as follows:

```
<xattr>
      <key>ltfs.mediaPool.name</key>
      <value>name</value>
</xattr>
```

Example:

```
<xattr>
      <key>ltfs.mediaPool.name</key>
      <value>Animation</value>
</xattr>
```

The UUID of the media pool to which an LTFS volume belongs may be stored in an extended attribute for the root directory element in an index (see 9.2). as follows:

```
<xattr>
      <key>ltfs.mediaPool.uuid</key>
      <value>uuid</value>
</xattr>
```

Example:

```
<xattr>
      <key>ltfs.mediaPool.uuid</key>
      <value>8c55c141-8cff-4312-b4ec-b18a335f495d</value>
</xattr>
```

F.4.1.1 Media Pool MAM Attributes

The name of the media pool to which a volume belongs may optionally be stored in the Media Pool MAM host type attribute (see Section 10.4 Use of Host-type Attributes for LTFS). If set, this attribute shall be left-aligned and null-terminated, and its value should be consistent with the value of the ltfs.mediaPool.name extended attribute for the root directory element in an index (see Section 9.2).

In addition, application-specific additional information may optionally be stored in the Media Pool MAM host type attribute, with or without the name of a media pool. If additional information is stored in the Media Pool MAM attribute, it must be enclosed in square brackets ("[" and "]"), and must immediately precede the null terminator for the attribute. For example, if the Media Pool MAM attribute contains both a pool name and additional information, the values are stored as:

media pool name[additional information]

If this attribute contains only additional information, it is stored as:

[additional information]

If the Media Pool MAM attribute does not contain any additional information, it should not contain the square bracket delimiters. Square brackets are reserved characters for this MAM attribute, and may not be used in the media pool name or additional information values. The additional information value, if any, is never stored in the LTFS index.

If the number of bytes required to store the media pool name (or the combination of the media pool name and the additional information) exceeds the available attribute storage size of 160 bytes, then the media pool name stored in the attribute shall be truncated at the most appropriate character boundary. For an example of this host type attribute, see Section 10.4.11 Example attributes.

Writing to the ltfs.mediaPool.name extended attribute updates the values of both the extended attribute in the LTFS index and the media pool name portion of the Media Pool MAM attribute. The additional information value of the Media Pool MAM attribute is written and read using the ltfs.mediaPool.additionalInfo extended attribute. Writing the ltfs.mediaPool.name extended attribute should attempt to preserve any existing additional information previously written to the Media Pool MAM attribute through the use of the ltfs.mediaPool.additionalInfo extended attribute.

The UUID of the media pool to which a volume belongs may optionally be stored in the Media Pool Globally Unique Identifier host type attribute (see Section 10.4 Use of Host-type Attributes for LTFS). This value should be consistent with the value of the ltfs.mediaPool.uuid extended attribute for the root directory element in an index (see Section 9.2).

Annex G (informative) Character representations

This annex describes how various characters are expected to be expressed in an LTFS index, both for implementations which comply with version 2.3 or later of this specification, and for implementations complying with previous versions.

Table G.1 — Character representations : version 2.3 or later

Hex	Character	File name, Directory name, Name pattern[4]	Symlink target name	Extended Attribute name
00	NUL	Not allowed		
01	SOH	Percent encode		
02	STX	Percent encode		
03	ETX	Percent encode		
04	EOT	Percent encode		
05	ENQ	Percent encode		
06	QCK	Percent encode		
07	BEL	Percent encode		
08	BS	Percent encode		
09	TAB	Allowed		
0A	NL	Allowed		
0B	VTAB	Percent encode		
0C	NP	Percent encode		
0D	CR	Allowed		
0E	SO	Percent encode		
0F	SI	Percent encode		
10	DLE	Percent encode		
11	DC1	Percent encode		
12	DC2	Percent encode		
13	DC3	Percent encode		
14	DC4	Percent encode		
15	NAK	Percent encode		
16	SYN	Percent encode		
17	ETB	Percent encode		
18	CAN	Percent encode		
19	EM	Percent encode		
1A	SUB	Percent encode		
1B	ESC	Percent encode		
1C	FS	Percent encode		
1D	GS	Percent encode		
1E	RS	Percent encode		
1F	US	Percent encode		
22	"	XML Entity replacement		
25	%	Percent encode or Allowed [1]		

Hex	Character	File name, Directory name, Name pattern[4]	Symlink target name	Extended Attribute name
26	&	XML Entity replacement		
27	'	XML Entity replacement		
2F	/	Not allowed	Allowed[3]	Allowed
3A	:	Percent encode		
3C	<	XML Entity replacement		
3E	>	XML Entity replacement		
All other characters[2]		Allowed		

NOTE 1 When writing an index, if the percent character is present in a name and no other characters in the name require percent-encoding, an implementation may choose whether to use the percent character as-is (without encoding) or to percent-encode it (i.e. represent as %25). When reading and parsing an index, an implementation should accept either and treat them as equivalent.

NOTE 2 Although other characters could be percent-encoded, it is strongly recommended that they should be used as-is (without encoding) to minimize backward compatibility issues with implementations conforming to earlier versions of the standard.

NOTE 3 The forward slash character is allowed in the symlink target name, but the name of the file which contains the symlink element follows the same rules as for other file & directory names (i.e. the forward slash character is not allowed).

NOTE 4 See Section 7.5 for the description of the name pattern format.

Table G.2 — Character representations : version 2.2 or earlier

Hex	Character	File name, Directory name, Name pattern	Symlink target name	Extended Attribute name
00	NUL	Not allowed		
01	SOH	Not allowed		
02	STX	Not allowed		
03	ETX	Not allowed		
04	EOT	Not allowed		
05	ENQ	Not allowed		
06	QCK	Not allowed		
07	BEL	Not allowed		
08	BS	Not allowed		
09	TAB	Allowed		
0A	NL	Allowed		
0B	VTAB	Not allowed		
0C	NP	Not allowed		
0D	CR	Allowed		
0E	SO	Not allowed		
0F	SI	Not allowed		
10	DLE	Not allowed		
11	DC1	Not allowed		

Hex	Character	File name, Directory name, Name pattern	Symlink target name	Extended Attribute name
12	DC2	Not allowed		
13	DC3	Not allowed		
14	DC4	Not allowed		
15	NAK	Not allowed		
16	SYN	Not allowed		
17	ETB	Not allowed		
18	CAN	Not allowed		
19	EM	Not allowed		
1A	SUB	Not allowed		
1B	ESC	Not allowed		
1C	FS	Not allowed		
1D	GS	Not allowed		
1E	RS	Not allowed		
1F	US	Not allowed		
22	"	XML Entity replacement		
25	%	Allowed		
26	&	XML Entity replacement		
27	'	XML Entity replacement		
2F	/	Not allowed		
3A	:	Not allowed		
3C	<	XML Entity replacement		
3E	>	XML Entity replacement		
All other characters		Allowed		

Annex H (informative) Incremental Indexes

This annex describes Incremental Indexes: the motivation behind them, their impact on compatibility with earlier versions of LTFS, hints for processing them, and whatever other information on the subject the authors consider useful. Some of this information is a restatement of information found elsewhere in this specification, and is included here in an attempt to collect much of the information relevant to Incremental Indexes in a single place.

H.1 Background

LTFS implementations periodically write copies of the file system's volatile state to the Data Partition of the tape, in the form of an LTFS Index. There are two major reasons for doing this: most importantly, to provide a recovery point for the file system in the case of a system failure, but also to provide a roll-back point for accessing an earlier state of the tape volume (either permanently or temporarily). As tape media has continued to grow in capacity, the number of files written to the tape volume has likewise tended to increase. Because any copy of the index written to tape has historically been required to contain the full state of the file system, the size of the index (and the time taken to write it) have also been growing, to the point where the overhead of indexes in the data partition has become problematic for some installations.

As a result, versions of LTFS claiming compliance with version 2.5.0 or later of the LTFS format specification are required to support Incremental Indexes. (We refer to a traditional LTFS index, which contains a full copy of the file system state, as a Full Index.) An Incremental Index contains only the information necessary to update the file system from its state at the previous index (either Full or Incremental) to its state at the time the Incremental Index is written; i.e., it contains only the changes to the file system since the previous index was written to tape.

Incremental Indexes are only written in the Data Partition, and are never written in order to create a consistent volume suitable for unmounting. Thus, the Index at the end of the Data Partition on a consistent volume must always be a Full Index (as, of course, must all of the indexes in the Index Partition).

Versions of LTFS claiming compliance with version 2.5.0 or later are not required to create Incremental Indexes (though their implementors are strongly encouraged to do so). However, such implementations are required to recognize and correctly process Incremental Indexes when they are encountered in a Data Partition.

H.2 Backwards Compatibility

Changing the format of an index on an LTFS volume has serious implications for backwards compatibility, specifically for the ability of an earlier implementation to read data from a later volume. Therefore Incremental Indexes have been merged into the LTFS format with the specific goal of minimizing their impact on backwards compatibility with earlier implementations.

Of course, Incremental Indexes do not completely replace traditional Full Indexes; they exist in the Data Partition along with Full Indexes. At the very least, an implementation must ensure that there are current Full Indexes in both the Data and Index Partitions every time the the volume is made consistent (ready for unmount).

NOTE: We expect an implementation to continue to write Full Indexes to the Data Partition periodically (perhaps after every 5-10 Incremental Indexes). The reason for this is the overhead of recovery (or roll-back) using Incremental Indexes: in order to recover to the point of an Incremental Index the LTFS system must locate the most recent Full Index prior to the desired Incremental Index, process that Full Index, then apply the changes from each subsequent Incremental Index until the desired Incremental Index has been processed.

Because a consistent volume must have Full Indexes both in the Index Partition and at the end of the Data Partition, mounting and processing a consistent volume containing Incremental Indexes will pose no problem for earlier LTFS implementations.

In earlier versions of the LTFS format the Index back pointer (**previousgenerationlocation**) forms an unbroken chain of back pointers to the first index written in the Data Partition. This chain is preserved in

the current version, with the added provision that the chain includes only Full Indexes. Thus, an earlier implementation of LTFS processing this back-chain will operate correctly, and will be able to access any Full Index in the chain. However, the earlier implementation will skip over all of the Incremental Indexes on the volume. As a result, an earlier implementation processing a volume with Incremental Indexes will find far fewer points to which it can roll back than will a current implementation.

Depending on how recovery processing is implemented, an earlier implementation attempting to recover an inconsistent volume containing Incremental Indexes will likely have one of two outcomes: it may fail due to not immediately finding a recognizable (Full) index where it expects one, or it may succeed by continuing to search the tape volume until it encounters a recognizable index. In the latter case, while recovery will succeed, there will be a loss of data due to the recovery process skipping Incremental Indexes which it does not recognize.

Neither of these is an ideal outcome. In practice, however, this is unlikely to be a problem, as it is improbable that an inconsistent volume would be moved to an environment running an earlier version of LTFS before being recovered. (It may also be possible to use the latest version of an LTFS recovery utility, e.g. ltfsck, in an environment that continues to run an earlier implementation of LTFS; this would allow proper recovery of volumes containing Incremental Indexes even in such an environment.)

In summary, the steps taken to mitigate backwards compatibility issues make it probable that earlier LTFS implementations will be able to process volumes containing Incremental Indexes correctly and without issues.

H.3 Traversing the Index Back Pointer Chain

During rollback and potentially during recovery processing it may be necessary to follow the index back chain to a desired index. In earlier versions of LTFS this was done by simply following the **previousgenerationlocation** element pointer. However, version 2.5.0 of the LTFS format specification introduces a second index back pointer, **previousincrementallocation**.

NOTE: Every index (except the first one written in the Data Partition) will continue to contain a **previousgenerationlocation** element, and that element will always point to the preceding Full Index.

This **previousincrementallocation** element may occur in either a Full or Incremental Index, and if present, it indicates that the index written immediately before the one in which it occurs was an Incremental Index and provides a pointer to it. If this element exists, it should be used as the index back chain pointer in order to traverse all of the indexes in the chain. Only when this element does not exist (or when, for some reason, it is necessary to skip over Incremental Indexes) should the **previousgenerationlocation** pointer be used. (As noted in Section H.2, earlier implementations of LTFS will only recognize the **previousgenerationlocation** element, and thus will only follow the Full Index chain.)

Refer to Figure 7 in Section 5.4.3 Back Pointer for an illustration of index back chains with Incremental Indexes.

H.4 Incremental Index Format

The format of an Incremental Index has been kept as similar to the format of a Full Index as possible. With the exception of the change of the **ltfsindex** element to **ltfsincrementalindex**, the only new element is **deleted**. The **deleted** element indicates that the file or directory should be deleted from the index, and in the case of a directory, that the entire directory sub-tree is truncated (all child objects are also deleted). (This is consistent with file system behavior, which does not allow the deletion of non-empty directories.)

Although the XML elements used in Full and Incremental Indexes are almost identical, the rules governing the use of the XML elements are quite different. Because a Full Index must completely describe the state of the LTFS file system, all of the XML elements needed to represent the state of every object (file or directory) in the file system are required in a Full Index.

An Incremental Index is only required to describe the changes from the prior index. At a high level, there are three cases for what XML is required to be included in an Incremental Index:

- Deleted objects: Within the object element (**directory** or **file**), the only elements that can appear are **name** and **deleted**.
- New objects: All of the information that would have been recorded in a Full Index for the new object must appear in the Incremental Index.
- Modified objects: Only the **name** and **fileuid** elements are required. Any XML elements needed to describe the changed attributes of the object since the prior index was written must also be included.

However, there are some additional details regarding the Incremental Index XML that are important to understand:

- Because the Incremental Index is recorded in the same tree structure as a Full Index, all **directory** entries that are necessary to navigate to the correct location in the name space must also be included in the Incremental Index. These unmodified **directory** elements should contain only the **name** and **contents** elements necessary for navigating the name space, and are the only case where an unmodified object must be included in an Incremental Index.
- In the prior discussion, objects are described as either deleted, new, or modified; those definitions are from a file system name space point of view. When an object is moved (or renamed) from one location in the name space to another, we tend not to think of it as a new object; it has the same attributes, including its fileuid, as before it was moved. From a name space view, though, the object has been deleted from its original location, and has been added to its new location. An Incremental Index must represent it in that way: it must be deleted from its old location, and added as a new object in its new location. For a file, this is simply one **file** element with a **name** and **deleted** element, and another **file** element with all of the information for the file, including the original fileuid. For a directory, this is also a single **directory** element with a **name** and **deleted** element, however the new **directory** element must contain all of the information for the entire name space subtree that was moved. (Admittedly, this could be quite large, but no larger than what would have been recorded in a Full Index).
- For modified objects, ideally the Incremental Index would contain only those elements whose value has actually changed since the prior index. Practically speaking, this goal may be impossible, since it could require having some type of “change bit” for each possible value in the index. Implementations should strive to write as little information as is practical for modified objects; however, writing more information than is actually needed is harmless, and impacts only the size of the Incremental Index.
- In order to avoid ambiguity, the **extentinfo** and **extendedattributes** elements, if included in an Incremental Index, must occur in their entirety.
- Only a single entry for a given name in the name space should appear in the index. Thus, if object “foo” is deleted and a new object “foo” (with a different fileuid) is created in its place, only the new entry for the object should be included in the Incremental Index. (An object with the same name as in the prior index but a different fileuid implies deletion of the prior object and addition of the new one; refer to the processing shown in Figure H.1.)
- A minimal (i.e., empty) Incremental Index must contain (besides the required Preface section) the **directory** element for the volume root directory, and a **name** and an empty **contents** element for that directory. (As in a Full Index, the **name** element contains the LTFS volume name). This is illustrated in the following XML document for an empty Incremental Index (omitting the Preface):

```
<?xml version="1.0" encoding="UTF-8"?>
<ltfsincrementalindex version="2.5.0">
...
   <directory>
      <name>LTFS Volume Name</name>
      <contents></contents>
   </directory>
</ltfsincrementalindex>
```

As a further example, the following XML document shows an Incremental Index which records the addition of a second data block to an existing file **file_1** in directory **dir_B**, which is itself a child of the directory **dir_A** located at the root of the volume. All other contents of the volume are unchanged.

```
<?xml version="1.0" encoding="UTF-8"?>
<ltfsincrementalindex version="2.5.0">
...
   <directory>
      <name>LTFS Volume Name</name>
      <contents>
         <directory>
            <name>dir_A</name>
            <contents>
               <directory>
                  <name>dir_B</name>
                  <contents>
                     <file>
                        <fileuid>131</fileuid>
                        <name>file_1</name>
                        <length>512</length>
                        <modifytime>2019-01-23T16:24:31.228983707Z</modifytime>
                        <changetime>2019-01-23T16:24:31.228983707Z</changetime>
                        <extentinfo>
                           <extent>
                              <partition>b</partition>
                              <startblock>18</startblock>
                              <byteoffset>0</byteoffset>
                              <bytecount>256</bytecount>
                              <fileoffset>0</fileoffset>
                           </extent>
                           <extent>
                              <partition>b</partition>
                              <startblock>49</startblock>
                              <byteoffset>0</byteoffset>
                              <bytecount>256</bytecount>
                              <fileoffset>256</fileoffset>
                           </extent>
                        </extentinfo>
                     </file>
                  </contents>
               </directory>
            </contents>
         </directory>
      </contents>
   </directory>
</ltfsincrementalindex>
```

H.5 Processing Incremental Indexes

As has already been stated, versions of LTFS claiming compliance with version 2.5.0 or later of the LTFS

format specification must be able to process Incremental Indexes during recovery and roll-back processing. It has also been explained that this process must begin by processing the preceding Full Index, then successively applying changes from the chain of desired Incremental Indexes.

The format of the Incremental Index has been designed not only to minimize the space taken on tape, but also to make the application of the index to an existing file system state relatively straightforward.

The flowchart in Figure H.1 shows the high-level flow of Incremental Index application that was developed for a prototype implementation. In this figure, the term "dentry" refers to an entry in the file system metadata to which the index is to be applied (often in memory); the term "object" refers to either a **file** or **directory** element encountered in the Incremental Index. Processing starts by passing the "root" **directory** element of the LTFS volume (refer to Section 9.2.3 Required elements for every index) to the function which applies changes to the index. That function processes each of the objects passed to it in turn, deleting, adding, or updating files and directories as needed. After processing a directory that is not marked as having been deleted, it immediately descends recursively into that directory (creating a depth-first traversal of the Incremental Index tree).

Note: There is a case in the flowchart where an object indicating a deleted file or directory is found but there is no matching dentry. This may appear to be an error case, but can validly occur when an object is both created and deleted in the same index generation.

H.6 Miscellaneous

As noted earlier, it is expected that Incremental Indexes and Full Indexes will be intermixed in the Data Partition. We recommend that an implementation allow the maximum number of Incremental Indexes that can be written before writing a Full Index to be specified by the installation; we suggest that the implementation also set an upper limit on that value (such as 10). We also recommend that a value of zero be used to indicate that no Incremental Indexes are to be written, making version 2.5.0 volumes identical to (and completely interchangeable with) earlier implementations of LTFS.

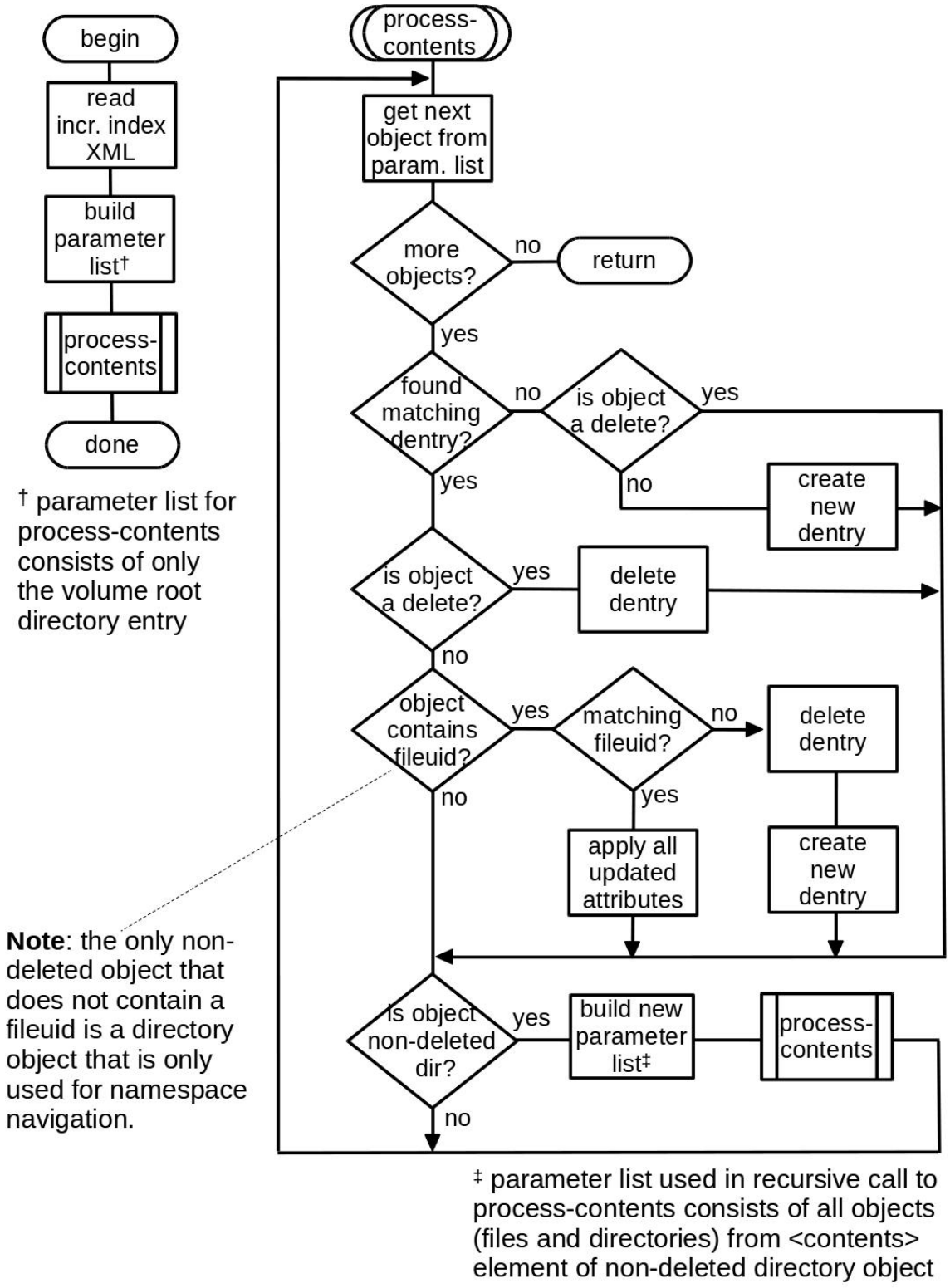


Figure H.1 — Processing an Incremental Index (flowchart)