Linear Tape File System (LTFS) Format Specification

August 17, 2011

LTFS Format Version 2.0.1

This document presents the requirements for an interchanged tape conforming to a self describing format. This format is used by the Linear Tape File System (LTFS). This document does not describe implementation of LTFS itself.

Table of Contents

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 meta data 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.

1.1 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.

1.2 Versions

This document describes version 2.0.1 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 the following table:

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: 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: 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 the following table.

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 $N_A < N_B$	$M_A.N_A.R_A$ 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.

1.3 Conformance

Recorded media claiming conformance to this format shall be in a consistent state when interchanged or stored. See 2.1.4 **Consistent State**.

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 6 **Label Format** and 7 **Index Format**.

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

2 Definitions and Acronyms

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

2.1 Definitions

2.1.1 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.

2.1.2 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.

2.1.3 Content Area

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

2.1.4 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 Index Construct in the Data Partition.

2.1.5 Data Extent

A contiguous sequence of recorded blocks.

2.1.6 Data Partition

An LTFS partition primarily used for data files.

2.1.7 File

A group of logically related extents together with associated file meta-data.

2.1.8 filesystem sync

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

2.1.9 generation number

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

2.1.10 Index

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

2.1.11 Index Construct

A data construct comprised of an Index and file marks.

2.1.12 Index Partition

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

2.1.13 Label Construct

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

2.1.14 Linear Tape File System (LTFS)

This document describes the Linear Tape File System Format.

2.1.15 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.

2.1.16 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 **A LTFS Label XML Schema**.

2.1.17 LTFS Partition

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

2.1.18 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 must have the same UUID. All LTFS partitions in an LTFS volume are *related partitions*.

2.1.19 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 in the T10/SPC-4 standard.

2.1.20 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.

2.1.21 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 ($0x00$).

2.1.22 UUID

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

2.1.23 Volume Change Reference (VCR)

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

2.2 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

3 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 must contain exactly one Data Partition and exactly one Index Partition.

3.1 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 the figure below.

The Label Construct is described in 3.2 **LTFS Constructs** and in 6 **Label Format**. The Content Area contains some number of interleaved Index Constructs and Data Extents. These constructs are described in 3.2 **LTFS Constructs** and 7 **Index Format**. The precise layout of the partitions is defined in 3.3 **Partition Layout**.

3.2 LTFS Constructs

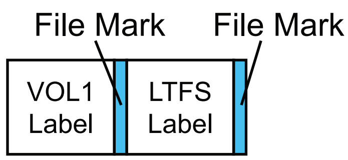
LTFS constructs are comprised of file marks and records. These are also known as 'logical objects' as found in T10 SSC specifications and are not described here. An LTFS volume contains three kinds of constructs.

- 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 meta-data 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.

3.2.1 Label Construct

Each partition in an LTFS volume shall contain a Label Construct with the following structure. As shown in the figure below, 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 must 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 6 **Label Format**.



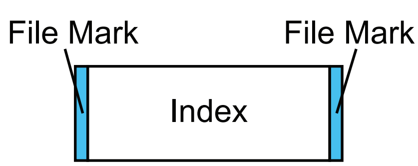
3.2.2 Data Extent

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 records within a Data Extent must 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 4 **Data Extents**.

3.2.3 Index Construct

The figure below 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 must 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 must 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.



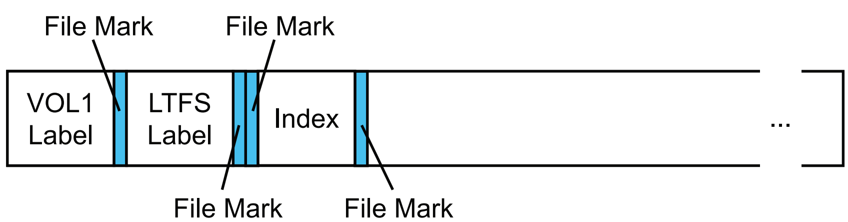
Indexes also include references to other Indexes in the volume. References to other Indexes are used to maintain consistency between partitions in a volume. These references (back pointers and self pointers) are described in 3.4 **Index Layout**.

The content of the Index is described in 7 **Index Format**.

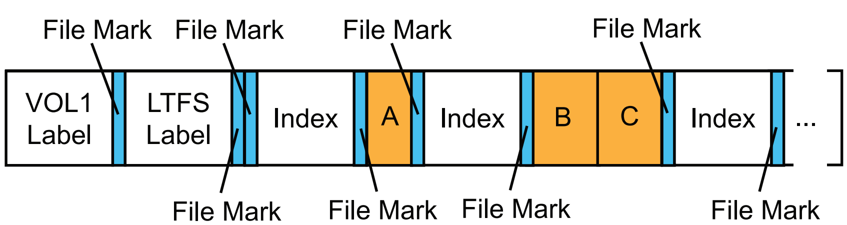
3.3 Partition Layout

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 must be an Index Construct.

The figure below illustrates an empty complete partition. It contains a Label Construct followed by an Index Construct. This is the simplest possible complete partition.



The figure below 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.



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.

3.4 Index Layout

Each Index data structure contains three pieces of 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 Index present on the Data Partition immediately before this Index was written.

3.4.1 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. 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 the following elements:
  - access time values for files and directories (described in 7.2 **Index**),
  - the self pointer (described below), and
  - the back pointer (described below).

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`'.

3.4.2 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

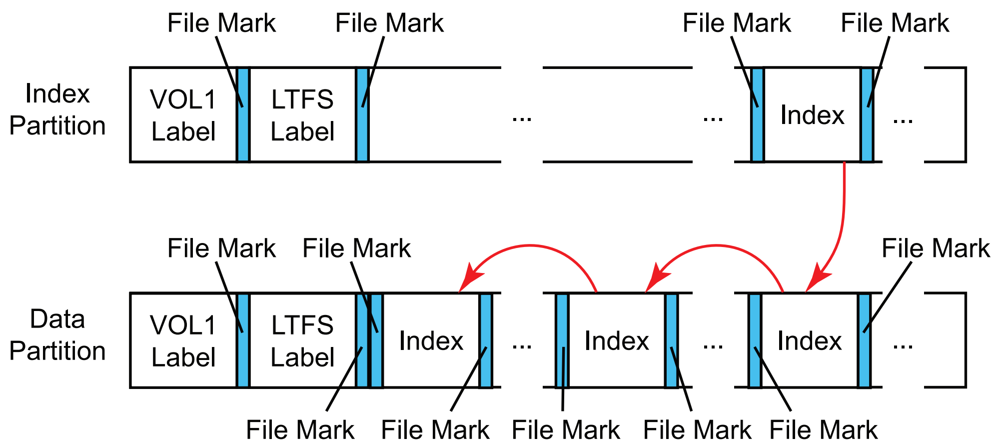
The self pointer is used to distinguish between Indexes and Data Extents. An otherwise valid Index with an invalid self pointer must be considered a Data Extent for the purpose of verifying that a volume is valid and consistent. This minimizes the likelihood of accidental confusion between a valid Index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

3.4.3 Back Pointer

Each Index contains at most one back pointer, defined as follows.

- If the Index resides in the Data Partition, the back pointer shall contain the block position of the preceding 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 7.2 **Index**.
- If the Index resides in the Index Partition and has generation number N then the back pointer for the Index shall contain either the block position of an Index having generation number N in the Data Partition, or the block position of the last 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 must contain a back pointer to the final index in the Data Partition.
- 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. The figure below illustrates this state.


File Mark
File Mark
File Mark
File Mark
Index Partition
VOL1 Label
LTFS Label
...
...
Index
...
File Mark
File Mark
File Mark
Data Partition
VOL1 Label
LTFS Label
Index
...
Index
...
Index
...
File Mark
File Mark
File Mark
File Mark
File Mark

4 Data Extents

A Data Extent is a set of one or more sequential records subject to the conditions listed in 3.2.2 **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.

4.1 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 must be strictly less than the size of the start block. The use of byte offset is described in section 4.2.3.
- **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 must 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.

Implementors are encouraged to record extents in the same logical order as the 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 must 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.

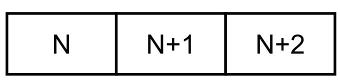
4.2 Extents Illustrated

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 below. Note that not all extents in an extent list must be on the same partition.

4.2.1 Starting and ending Data Extent with full block

The following figure illustrates an extent of 3 full size blocks contained within a Data Extent of 3 blocks, N through $N + 2$.



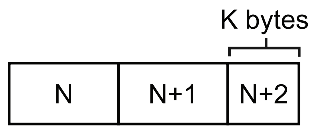
The extent list entry for this extent is:

Start Block	**Offset**	**Length**
N	0	$3 \times Blk$

Note: Blk is the length of a full sized block.

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

The following figure 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 $N+2$.



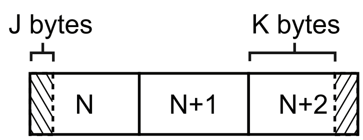
The extent list entry for this extent is:

Start Block	Offset	Length
N	0	$(2 \times Blk) + K$

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

4.2.3 Starting and ending Data Extent in mid-block

The following figure 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.



The extent list entry for this extent is:

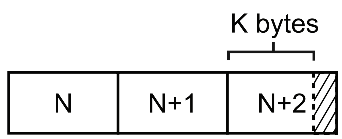
Start Block	Byte Offset	Byte Count
N	J	$(Blk - J) + Blk + K$

4.3 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.

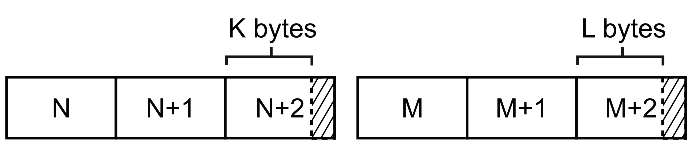
4.3.1 Simple Files

The following figure 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.



Start Block	Byte Offset	Byte Count	File Offset
N	0	$(2 \times Blk) + K$	0

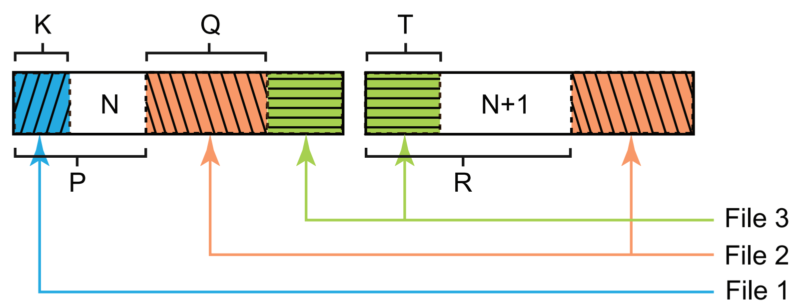
The following figure illustrates a file contained in two Data Extents of three blocks each. The data fills the first two blocks of extent N and K bytes of block $N+2$, and the first two blocks of extent M and L bytes of block $M+2$. The last block of each extent, block $N+2$ and $M+2$, may be fractional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



Start Block	Byte Offset	Byte Count	File Offset
N	0	$(2 \times Blk) + K$	0
M	0	$(2 \times Blk) + L$	$(2 \times Blk) + K$

4.3.2 Shared Blocks

The following figure 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 $N+1$.



The extent lists for files 1, 2, and 3 are:

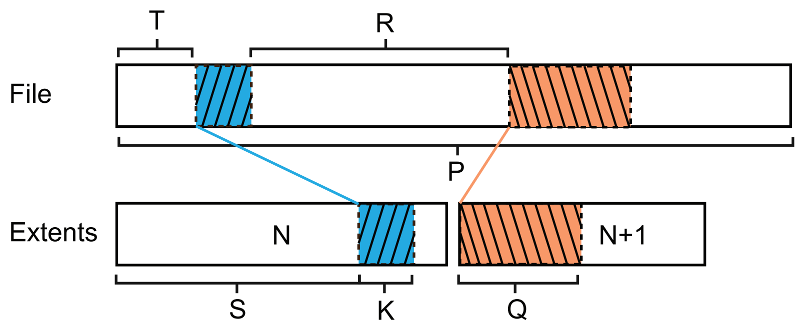
	Start Block	Byte Offset	Byte Count	File Offset
File 1	N	0	K	0
File 2	N	P	Q	0
	$N+1$	R	$Blk - R$	Q
File 3	N	$P+Q$	$Blk - P - Q + T$	0

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 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 $N+1$).

4.3.3 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, the all bytes not encoded in the extent list shall be treated as zero ($0x00$) bytes.

The following figure 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 ($0x00$). The file starts with T bytes with value zero($0x00$). 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 ($0x00$) 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 ($0x00$) to the defined file size P.



The extent list this file is:

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

Note: 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 compilance 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.

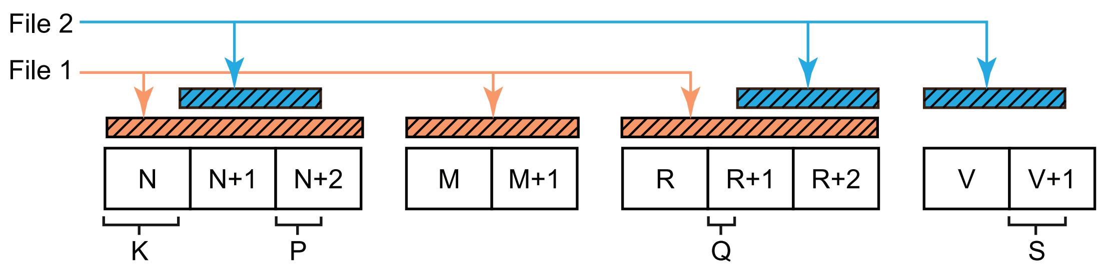
Note: Version 1.0 of this specification did not support sparse files.

4.3.4 Shared Data

The following figure 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 the following figure, 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 below. The two files share some of the data in blocks N, $N + 1$, $N + 2$, $R + 1$ and $R + 2$.



The extent lists for files 1 and 2 are:

	Start Block	**Byte Offset**	**Byte Count**	**File Offset**
File 1	N	0	$3 \times Blk$	0
	M	0	$2 \times Blk$	$3 \times Blk$
	R	0	$3 \times Blk$	$(3 \times Blk) + (2 \times Blk)$
File 2	N	K	$(Blk - K) + Blk + P$	0
	R+1	Q	$(Blk - Q) + Blk$	$(Blk - K) + Blk + P$
	V	0	$Blk + S$	$(Blk - K) + Blk + P + (Blk - Q) + Blk$

5 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.

5.1 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.

5.2 Creator format

LTFS creator values shall be recorded in conformance with the string format defined in 5.6 **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 the name of the executable that wrote the LTFS volume.

An example of the recommended content for creator values is shown below.

```
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 an space character, followed by an hyphen character, followed by another space character. The recommended content for the creator value is *`Company Product Version - Platform - binary name`* where:

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.

5.3 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 5.6 **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 must be ignored when decoding the string.

5.4 Name format

File and directory names, and extended attribute keys in an LTFS Volume must conform to the following naming rules.

Names must be valid Unicode and must be 255 code points or less after conversion to Normalization Form C (NFC). Names must 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 following.

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

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

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

5.5 Name pattern format

File name patterns in data placement policies shall be valid names as defined in 5.4 **Name format**. A file name pattern shall be compared to a file name using the following 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 a filenames 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.

5.6 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.

5.7 Time stamp format

Time stamps in LTFS data structures must be specified as a string conforming with the ISO 8601 date and time representation standard. The time stamp must be specified in UTC (Zulu) time as indicated by the 'Z' character in the example below. The time must be specified with a fractional second value that defines 9 decimal places after the period in the format.

`2010-02-01T18:35:47.866846222Z`

The general time format is `YYYY-MM-DDThh:mm:ss.nnnnnnnnnZ` where:

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 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 must be padded to the full width of the symbol using `0` characters. For example, an integer month value of '`2`' must be recorded as '`02`' to fill the width of the *MM* symbol in the general time format.

5.8 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.

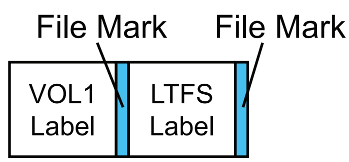
`30a91a08-daae-48d1-ae75-69804e61d2ea`

6 Label Format

This section describes the content of the Label Construct. The content of the Content Area is described in 3.2 **LTFS Constructs** and 7 **Index Format**.

6.1 Label Construct

Each partition in an LTFS Volume shall contain a Label Construct that conforms to the structure shown in the figure below. 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 must 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 must be recorded starting at the first logical block in the partition. Both Label constructs in an LTFS Volume must contain identical information with the exception of the "location" field in the XML data for the LTFS Label.



6.1.1 VOL1 Label

A VOL1 label recorded on an LTFS Volume shall always be recorded in a Label Construct as defined in 6.1 **Label Construct**.

The first record in a Label Construct is an ANSI VOL1 record. This record conforms to the ANSI Standard X 3.27. All bytes in the VOL1 record are stored as ASCII encoded characters. The record is exactly 80 bytes in length and has the following structure and content.

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

Note: 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.

6.1.2 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 Appendix A **LTFS Label XML Schema**. The LTFS Label shall be 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 6.1 **Label Construct**.

A complete schema for the LTFS Label XML data structure is provided in Appendix A **LTFS Label XML Schema**. An example LTFS Label is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<ltfslabel version="2.0.0">
    <creator>IBM LTFS 1.2.0 - Linux - mkltfs</creator>
    <formattime>2010-02-01T18:35:47.866846222Z</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 must be an XML data structure that conforms to the W3C Extensible Markup Language (XML) 1.0 standard. Every LTFS Label must have a first line that contains an XML Declaration as defined in the XML standard. The XML Declaration must 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 must contain the following information:

ltfslabel: this element defines the contained structure as an LTFS Label structure. The element must have a **version** attribute that defines the format version of the LTFS Label in use. This document describes LTFS Label version 2.0.1.

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 must contain the necessary information to uniquely identify the writer of the LTFS volume. The value must conform to the creator format definition shown in 5.2 **Creator format**.

formattime: this element must contain the time when the LTFS Volume was formatted. The value must conform to the format definition shown in 5.7 **Time stamp format**.

volumeuuid: this element must contain a universally unique identifier (UUID) value that uniquely identifies the LTFS Volume to which the LTFS Label is written. The **volumeuuid** element must conform to the format definition shown in 5.8 **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 must 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 must contain exactly one **index** element for the Index Partition and exactly one **data** element for the Data Partition, formatted as shown. A partition must exist in the LTFS Volume with a partition identifier that matches the identifier recorded in the **index** element. Similarly, a partition must 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 must 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 3.2.2 **Data Extent** and 4 **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 5.1 **Boolean format**. When the **compression** element is set, compression must be enabled when writing to the LTFS Volume. When the **compression** element is unset, compression must 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.

6.1.3 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.

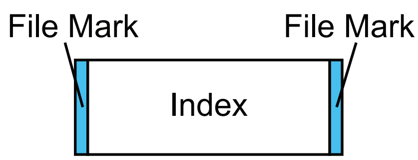
7 Index Format

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 decribed in 6 **Label Format**. Data Extents are described in 4 **Data Extents**.

7.1 Index Construct

Each Content Area in an LTFS Volume shall contain some number of Index Constructs that conform to the structure shown in the figure below. 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 must not be any additional data trailing the end of the Index.

The contents of the Index are defined in 7.2 **Index** below.



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

7.2 Index

An Index is an XML data structure that describes all data files, directory information and associated meta-data 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 7.1 **Index Construct**.

The LTFS Index shall conform to the Index XML schema provided in Appendix B **LTFS Index XML Schema**. The Index shall be be encoded using UTF-8 NFC.

A complete schema for the Index XML data structure is provided in Appendix B **LTFS Index XML Schema**. The remainder of this section describes the content of the Index using an example XML Index.

An Index consists of 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 may contain zero or more **directory** elements and zero or more **file** elements.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<ltfsindex version="2.0.0">
    <creator>IBM LTFS 1.2.0 - Linux - ltfs</creator>
    <volumeuuid>30a91a08-daae-48d1-ae75-69804e61d2ea</volumeuuid>
    <generationnumber>3</generationnumber>
    <comment>A sample LTFS Index</comment>
    <updatetime>2010-01-28T19:39:57.245954278Z</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>
    <higestfileuid>4</higestfileuid>
    <directory>
    ...
    </directory>
</ltfsindex>
```

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

Every Index must contain the following elements:

ltfsindex: this element defines the contained structure as an Index structure. The element must have a **version** attribute that defines the format version of the LTFS Index in use. This document describes LTFS Index version 2.0.1.

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 1.2 **Versions**.

creator: this element must contain the necessary information to uniquely identify the writer of the Index. The value must conform to the creator format definition shown in 5.2 **Creator format**.

volumeuuid: this element must 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 must conform to the format definition shown in 5.8 **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** must conform to the definitions provided in 3.4.1 **Generation Number**.

updatetime: this element shall contain the date and time when the Index was modified. The value must conform to the format definition shown in 5.7 **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 5.1 **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. Additional rules for the **allowpolicyupdate** element are provided in 7.2.3 **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 ($0x0$).

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 ($0x0$), an implementation may increment the **highestfileuid** value to create unique **fileuid** values for new directory and file entries.

A **highestfileuid** element value of zero ($0x0$) 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.

Every Index may contain the following elements:

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 5.6 **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 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 Index is recorded. The **startblock** element shall specify the first logical block number, within the partition, in which the back pointed 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**. All data values recorded in the **previousgenerationlocation** element must conform to the definitions provided in 3.4 **Index Layout**.

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 5.5 **Name pattern format**. A description of the rules associated with the **dataplacementpolicy** element is provided in 7.2.3 **Data Placement Policy**.

An example Index that omits the Preface section of the Index is shown below. 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="2.0.0">
    ...
    <directory>
        <fileuid>1</fileuid>
        <name>LTFS Volume Name</name>
        <creationtime>2010-01-28T19:39:50.715656751Z</creationtime>
        <changetime>2010-01-28T19:39:55.231540960Z</changetime>
        <modifytime>2010-01-28T19:39:55.231540960Z</modifytime>
        <accesstime>2010-01-28T19:39:50.715656751Z</accesstime>
        <backuptime>2010-01-28T19:39:50.715656751Z</backuptime>
        <contents>
            <directory>
                <fileuid>2</fileuid>
                <name>directory1</name>
                <creationtime>2010-01-28T19:39:50.740812831Z</creationtime>
                <changetime>2010-01-28T19:39:56.238128620Z</changetime>
                <modifytime>2010-01-28T19:39:54.228983707Z</modifytime>
                <accesstime>2010-01-28T19:39:50.740812831Z</accesstime>
                <backuptime>2010-01-28T19:39:50.740812831Z</backuptime>
                <readonly>false</readonly>
                <contents>
                    <directory>
                        <fileuid>3</fileuid>
                        <name>subdir1</name>
                        <readonly>false</readonly>
                        <creationtime>2010-01-28T19:39:54.228983707Z</creationtime>
                        <changetime>2010-01-28T19:39:54.228983707Z</changetime>
                        <modifytime>2010-01-28T19:39:54.228983707Z</modifytime>
                        <accesstime>2010-01-28T19:39:54.228983707Z</accesstime>
                        <backuptime>2010-01-28T19:39:54.228983707Z</backuptime>
                    </directory>
                </contents>
            </directory>
            <file>
                <fileuid>4</fileuid>
                <name>testfile.txt</name>
                <length>5</length>
                <creationtime>2010-01-28T19:39:51.744583047Z</creationtime>
                <changetime>2010-01-28T19:39:57.245291730Z</changetime>
                <modifytime>2010-01-28T19:39:57.245291730Z</modifytime>
                <accesstime>2010-01-28T19:39:57.240774456Z</accesstime>
                <backuptime>2010-01-29T20:21:45.424385077Z</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>
```

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) must contain the following information:

fileuid: this element must 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 ($0x1$).

name: this element must contain the name of the directory. A directory name must conform to the format specified in 5.4 **Name format**.

creationtime: this element must contain the date and time when the directory was created in the LTFS Volume. The value must conform to the format definition shown in 5.7 **Time stamp format**.

changetime: this element must contain the date and time when the extended attributes or readonly element for the directory was last altered. The value must conform to the format definition shown in 5.7 **Time stamp format**.

modifytime: this element must contain the date and time when the content of the directory was most recently altered. The value must conform to the format definition shown in 5.7 **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 must conform to the format definition shown in 5.7 **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 must conform to the format definition shown in 5.7 **Time stamp format**.

readonly: this element shall contain a value conforming to the boolean format definition provided in 5.1 **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.

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 7.2.1 **`extendedattributes` elements**. A **directory** element may have zero or one **extendedattributes** elements.

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.

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

fileuid: this element must 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 defined above as 1. All 'fileuid' elements must be unique in the index therefore no file may have a 'fileuid' less than 2.

name: this element must contain the name of the file. A file name must conform to the format specified in 5.4 **Name format**.

length: this element must contain the integer length of the file. The length is measured in bytes.

creationtime: this element must contain the date and time when the file was created in the LTFS Volume. The value must conform to the format definition shown in 5.7 **Time stamp format**.

changetime: this element must contain the date and time when the extended attributes or readonly element for the file was last altered. The value must conform to the format definition shown in 5.7 **Time stamp format**.

modifytime: this element must contain the date and time when the content of the file was most recently altered. The value must conform to the format definition shown in 5.7 **Time stamp format**.

accesstime: this element may contain the date and time when the content of the file 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 must conform to the format definition shown in 5.7 **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 must conform to the format definition shown in 5.7 **Time stamp format**.

readonly: this element shall contain a value conforming to the boolean format definition provided in 5.1 **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.

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 7.2.1 `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.

Every **extent** element shall describe the location where a file extent is recorded in the LTFS Volume. Every **extent** element must 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 must conform to the definitions provided in 3.2.2 **Data Extent** and 4 **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 4 **Data Extents** and 4.1 **Extent Lists**.

7.2.1 `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 below 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>2010-01-28T19:39:50.740812831Z</creationtime>
    <changetime>2010-01-28T19:39:56.238128620Z</changetime>
    <modifytime>2010-01-28T19:39:54.228983707Z</modifytime>
    <accesstime>2010-01-28T19:39:50.740812831Z</accesstime>
    <backuptime>2010-01-28T19:39:50.740812831Z</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 must 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 must conform to the format specified in 5.4 **Name format**. Extended attribute names must 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 5.3 **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 C **Reserved Extended Attribute definitions**.

7.2.2 Managing LTFS Indexes

An Index is a snapshot representation of the entire content of the LTFS Volume at a given point in time. Any alteration of an LTFS Volume shall record a complete snapshot of the entire content of the LTFS Volume.

Note: In practice, to maintain this snapshot sematic, an implementor generally should read the current Index from an LTFS Volume, make necessary changes to the Index and write the modified Index back to the LTFS Volume.

Implementations should handle additional unknown XML tags when they occur as children of the **ltfsindex**, **directory**, and **file** elements. These additional tags must 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.

7.2.3 Data Placement Policy

An 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 Parition.

An example Index that shows the elements that define the Data Placement Policy for an LTFS Volume is shown below. 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="2.0.0">
    ...
    <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 an Index. An Index may contain zero or one **dataplacementpolicy** elements.

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

Every **indexpartitioncriteria** element must 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 5.5 **Name pattern format**.

7.2.4 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 current Index as described in 7.2.3 **Data Placement Policy**.

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

7.2.4.1 Allow Policy Update is set

If the current Allow Policy Update value is set, as defined in 7.2.3 **Data Placement Policy**, a writer may record an Index that indicates the Allow Policy Update value is set or unset.

If the current Allow Policy Update value is set, as defined in 7.2.3 **Data Placement Policy**, a writer may record an 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 7.2.3 **Data Placement Policy**, a writer may record an 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 7.2.3 **Data Placement Policy**, a writer may record an Index without any **dataplacementpolicy** element.

7.2.4.2 Allow Policy Update is unset

If the current Allow Policy Update value is unset, as defined in 7.2.3 **Data Placement Policy**, a writer shall only record an Index that indicates the Allow Policy Update is unset.

If the current Allow Policy Update value is unset, as defined in 7.2.3 **Data Placement Policy**, a writer shall only record an 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 7.2.3 **Data Placement Policy**, a writer shall only record an Index with **dataplacementpolicy** values when those values exactly match the **dataplacementpolicy** values recorded in the previous generation of the Index.

7.2.5 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 Index as described in 7.2.3 **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 7.2.3 **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 5.5 **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 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 5.5 **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 7.2 **Index**, every 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.

8 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. Values stored in the MAM are stored on the medium 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 implementors use the MAM attributes as described in 8.3 **Use of Volume Coherency Information for LTFS** if such usage is supported by the underlying storage technology.

Note: For consistency with the referenced specifications, throughout the Medium Auxiliary Memory section the word 'Volume' is used to refer to a data storage medium (e.g. a tape cartridge). The words 'LTFS Volume' will is used when referencing an 'LTFS Volume' as defined in 2.1.18 LTFS Volume and throughout this document.

8.1 Volume Change Reference

Volume Change Reference (VCR) is a non-repeating, unique value associated with a volume coherency point. The following is for information only. 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.

8.2 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 the table below. 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 the following table.

The following is for information only. See the T10/SSC-4 Standard for a complete description of the Volume Coherency Information attribute.

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

- VCR Length: this field contains the length of the VCR field. The VCR Length field is a one-byte field.
- 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.
- 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.
- 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.
- 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.
- 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.

8.3 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 the following table. All offsets and lengths are measured in bytes.

Offset	**Length**	**Value**	**Notes**
0	4	‘`LTFS`’	
4	1	$0x00$	string terminator (binary)
5	36	<volume UUID>	as defined in 5.8 **UUID format**
41	1	$0x00$	string terminator (binary)
42	1	$0x01$	version number (binary)

Note: Single quotation marks in the ‘Value’ column above 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 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 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.

9 Certification

TBD

Voluntary certification of compliance to the LTFS can be obtained by submitting Volume images to ...

Certified compliance allows the display of the "LTFS Compliant" logo.

Appendices

A 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="2.0.0"/>
    </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>
```

B LTFS 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="allowpolicyupdate" type="xs:boolean"/>
        <xs:element name="dataplacementpolicy" type="policy" 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="xs:string"/>
        <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="xs:string"/>
```

```
        <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="extentinfo" minOccurs="0">
          <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: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="xs:string"/>
              <xs:element name="value">
                <xs:complexType mixed="true">
                  <xs:attribute name="type">
                    <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:complexType name="policy">
    <xs:sequence>
      <xs:element name="indexpartitioncriteria">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="size" type="xs:nonNegativeInteger"/>
            <xs:element name="name" type="xs:string" 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:simpleType name="version">
    <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]+\.[0-9]+\.[0-9]+"/>
      <xs:enumeration value="2.0.0"/>
    </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>
```

C 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. (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 list below. This section describes meaning of defined, reserved extended attributes.

Support for each of these defined, reserved extended attributes is optional for implementations in compliance with the LTFS Format Specification.

C.1 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.*

C.2 Volume Metadata

Extended Attribute	Value description
`ltfs.commitMessage`	*Commit message for the last Index on the LTFS Volume.*
`ltfs.indexCreator`	*Creator string for the Index. This string provides a human-readable identifier for the product that generated the Index. As defined in 5.2* ***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 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`' indicates that the previous Index starts at logical block* 55 *on partition `b`.*
`ltfs.indexTime`	*Date and time of when last LTFS Index was written to media.*
`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 5.2* ***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`	*Writing any value to this extended attribute shall trigger a filesystem sync on any implementation that supports this extended attribute. A filesystem sync is an operation that causes all in-memory filesystem changes to be flushed to the storage medium. The sync operation is not required to produce a consistent LTFS Volume. The sync operation must 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.*

Extended Attribute	**Value description**
`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.*

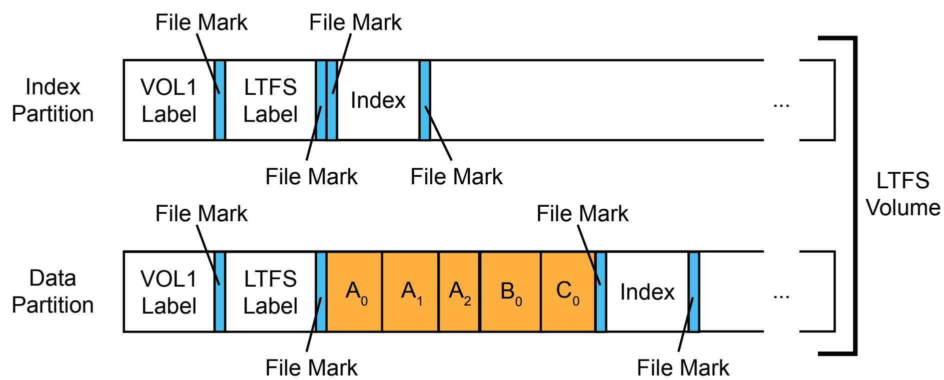
C.3 Media Metadata

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 writting 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.

Extended Attribute	**Value description**
`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 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 occured 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 occured 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. If the storage hardware cannot report this data the value will be the string "****UNKNOWN****".*

D Example of Valid Simple Complete LTFS Volume

The following figure 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.



E Complete Example LTFS Index

The following Index shows the important features of the Index format. Notes:

- 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 4.1 **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 4.3.4 **Shared Data**.

```
<?xml version="1.0" encoding="UTF-8"?>
<ltfsindex version="2.0.0">
    <creator>IBM LTFS 1.2.0 - Linux - ltfs</creator>
    <volumeuuid>5d217f76-53e6-4d6f-91d1-c4213d94a742</volumeuuid>
    <generationnumber>3</generationnumber>
    <updatetime>2010-02-16T19:13:49.532656726Z</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>8</highestfileuid>
    <directory>
        <fileuid>1</fileuid>
        <name>LTFS Volume Name</name>
        <readonly>false</readonly>
        <creationtime>2010-02-16T19:13:42.986549106Z</creationtime>
        <changetime>2010-02-16T19:13:47.517309274Z</changetime>
        <modifytime>2010-02-16T19:13:47.517309274Z</modifytime>
        <accesstime>2010-02-16T19:13:42.986549106Z</accesstime>
        <backuptime>2010-02-16T19:13:42.986549106Z</backuptime>
        <contents>
            <directory>
                <fileuid>2</fileuid>
```

```
        <name>directory1</name>
        <readonly>false</readonly>
        <creationtime>2010-02-16T19:13:43.006599071Z</creationtime>
        <changetime>2010-02-16T19:13:48.524075283Z</changetime>
        <modifytime>2010-02-16T19:13:46.514736591Z</modifytime>
        <accesstime>2010-02-16T19:13:43.006599071Z</accesstime>
        <backuptime>2010-02-16T19:13:43.006599071Z</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>2010-02-16T19:13:46.514736591Z</creationtime>
                <changetime>2010-02-16T19:13:46.514736591Z</changetime>
                <modifytime>2010-02-16T19:13:46.514736591Z</modifytime>
                <accesstime>2010-02-16T19:13:46.514736591Z</accesstime>
                <backuptime>2010-02-16T19:13:46.514736591Z</backuptime>
            </directory>
        </contents>
    </directory>
    <directory>
        <fileuid>4</fileuid>
        <name>directory2</name>
        <readonly>false</readonly>
        <creationtime>2010-02-16T19:13:43.007872849Z</creationtime>
        <changetime>2010-02-16T19:13:46.512350773Z</changetime>
        <modifytime>2010-02-16T19:13:46.512350773Z</modifytime>
        <accesstime>2010-02-16T19:13:43.007872849Z</accesstime>
        <backuptime>2010-02-16T19:13:43.007872849Z</backuptime>
        <contents>
            <file>
                <fileuid>5</fileuid>
                <name>sparse_file.bin</name>
                <length>20000000</length>
                <readonly>false</readonly>
                <creationtime>2010-02-16T19:13:45.012828533Z</creationtime>
                <changetime>2010-02-16T19:13:46.509553802Z</changetime>
                <modifytime>2010-02-16T19:13:46.509553802Z</modifytime>
```

```
            <accesstime>2010-02-16T19:13:45.012828533Z</accesstime>
            <backuptime>2010-02-17T19:15:34.032137221Z</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>2010-02-16T19:13:46.512350773Z</creationtime>
            <changetime>2010-02-16T19:13:46.513510263Z</changetime>
            <modifytime>2010-02-16T19:13:46.513510263Z</modifytime>
            <accesstime>2010-02-16T19:13:46.000000000Z</accesstime>
            <backuptime>2010-02-16T19:13:46.512350773Z</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>2010-02-16T19:13:44.009581288Z</creationtime>
                <changetime>2010-02-16T19:13:49.532111261Z</changetime>
                <modifytime>2010-02-16T19:13:49.532111261Z</modifytime>
                <accesstime>2010-02-16T19:13:49.527726902Z</accesstime>
                <backuptime>2010-02-16T19:13:44.009581288Z</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>2010-02-16T19:13:47.517309274Z</creationtime>
                <changetime>2010-02-16T19:13:47.519534438Z</changetime>
                <modifytime>2010-02-16T19:13:47.000000000Z</modifytime>
                <accesstime>2010-02-16T19:13:47.000000000Z</accesstime>
                <backuptime>2010-02-16T19:13:47.517309274Z</backuptime>
                <extendedattributes>
                    <xattr>
                        <key>author_name</key>
                        <value>Brian Biskeborn</value>
                    </xattr>
                </extendedattributes>
            </file>
        </contents>
    </directory>
</ltfsindex>
```

F Changes between Format Specification versions

This appendix contains a log of all high-level changes that were applied between each revision of this LTFS Format Specification document. Appendix lists changes in order of increasing specification version number to maintain continuity of sub-section numbering.

F.1 Changes between v1.0 and v2.0.0

Incremented specification version number to 2.0.1.

Updated specification date to August 17, 2011.

Improvements in specification text to remove ambiguity and clarify intention of the specification. These changes were made at several locations throughout the document.

Improvements to clarify description of MAM parameters in section 8 **Medium Auxiliary Memory**.

Removed reference to a specific version of the Unicode standard in 5.5 **Name pattern format**. This removes any requirement to use specific versions of Unicode support code in an implementation.

Improved description of *Name pattern format* to remove ambiguity in 5.5 **Name pattern format**.

Added description of LTFS Format specification version numbering in 1.2 **Versions**.

Updated XML Schema for Label and Index to match version number format in A **LTFS Label XML Schema** and B **LTFS Index XML Schema**.

Added specification of minimum and recommended blocksize value for LTFS Volumes to 6.1.2 **LTFS Label**.

Added definition of allowed version numbers to 6.1.2 **LTFS Label** and 7.2 **Index**.

Added definition of `fileoffset` tag in 7.2 **Index**.

Extended description in 4 **Data Extents** to support addition of `fileoffset` tag and associated functionality.

Added definition of `highestfileuid` tag in 7.2 **Index**.

Added definition of `fileuid` tag in 7.2 **Index**.

Added definition of `backuptime` tag in 7.2 **Index**.

Incremented version number in Application Client Specific Information (ACSI) structure shown in 8.3 **Use of Volume Coherency Information for LTFS**. This increment allows identification of LTFS Volumes written with a LTFS v1.0 compliant implementation. A widely used v1.0 implementation wrote ambiguous ACSI values due to an implementation bug.

Added definition of extended attributes in the `ltfs.*` namespace in Appendix C **Reserved Extended Attribute definitions**.

Added description for handling unknown XML tags in Index to 7.2.2 **Managing LTFS Indexes**.

F.2 Changes between v2.0.0 and v2.0.1

Incremented specification version number to 2.0.1.

Updated specification date to August 17, 2011.

Expanded F **Changes between Format Specification versions** to provide historical record of changes between revisions of LTFS Format Specification.

Improved description of constraints for two Indexes having the same generation number in 3.4.1 **Generation Number** to make it clear that differences in access time values is permitted between Indexes that are otherwise except for self pointer and index pointer values.

Added note in 3.4.1 **Generation Number** to explicitly state that Index generation numbers may increase by integer values other than 1.

Expanded description of the `ltfs.sync` extended attribute in Appendix C **Reserved Extended Attribute definitions**. The expanded description explicitly states that this extended attribute triggers a sync of the in-memory data to the storage media. That is, the operation is analogous to a POSIX sync operation.