INTERNATIONAL STANDARD

ISO/IEC 23001-14

First edition
2019-01

Information technology — MPEG systems technologies —

Part 14: Partial file format

Technologies de l'information — Technologies des systèmes MPEG —

Partie 14: Format de fichier partiel

Reference number
ISO/IEC 23001-14:2019(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

A list of all parts in the ISO 23001 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

The partial file format is designed to contain files partially received over a lossy link (with unreceived or corrupted sections), for further processing such as playback or repair. The file structure is object-oriented; a file can be decomposed into constituent objects very simply, and the structure of the objects inferred directly from their type. Files conforming to this format may be exchanged without losing information on losses, allowing redistribution (local gateways) or further repair processes.

The partial file format may be used to document reception of files, regardless of their bitstream format. For generic cases, it provides ways for file readers to resynchronize their parsing in case of byte losses. For cases where the documented file derives from ISO/IEC 14496-12, the partial file format provides additional tools, such as an index of the source file structures and data integrity information.

Annex A of this document defines the associated MIME type for files conformant to this document.

Information technology — MPEG systems technologies —

Part 14:
Partial file format

1 Scope

This document specifies the partial file format, which is a generic format for describing file partially received over lossy communication channels. This format contains the correctly received data, missing block identification, and repair information such as location of the file or high-level original indexing information. This format can be used with any file formats, and provides additional helper tools for formats deriving from ISO/IEC 14496-12.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 14496-12, *Information technology — Coding of audio-visual objects — Part 12: ISO Base Media file format*

IETF RFC 5905, *Network Time Protocol Version 4: Protocol and Algorithms Specification, Mills, D., et al, June 2010*

3 Terms, definitions, symbols, abbreviated terms and conventions

3.1 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1.1
chunk
addressable set of bytes of a partial segment

3.1.2
indexed box
box in the *source file* (3.1.5) indexed by a `BoxIndexBox` in the *partial file* (3.1.3)

3.1.3
partial file
name of the files conforming to the file format described in this document

3.1.4
partial segment
portion of the *partial file* (3.1.3) containing a single `PartialSegmentBox` and identifying its associated source data

3.1.5
source file
file described and encapsulated by the *partial file* (3.1.3)

3.2 Abbreviated terms

For the purposes of this document, the following abbreviated terms apply.

MD5	Message Digest 5
FDT	File Delivery Table
FEC	Forward Error Correction
FLUTE	File Delivery over Unidirectional Transport
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
RFC	Request for Comments
SHA	Secure Hash Algorithm

4 Partial file organization

4.1 Object structure

A partially received file, hereinafter referred to as partial file, is represented as a sequence of partial segments, preceded by a header. The header and the partial segments are formed as a series of objects, called boxes, as defined in ISO/IEC 14496-12; files under this document are object-structured files as defined in ISO/IEC 14496-12.

All object-structured files conformant to Clause 4 shall contain a `FileTypeBox` as defined in ISO/IEC 14496-12. The `FileTypeBox` shall contain a compatible brand of type `'paff'` for partial files, and of type `'pmff'` for mixed partial files.

In this document, top-level boxes (boxes not contained in other boxes) are indicated as being at ‘file’ level, with the notation “Container: File”.

The MIME type associated with a file conforming to this specification shall be formatted as defined in Annex A.

4.2 Design consideration

4.2.1 Features

The partial file format is intended to serve as a storage and exchange format for other file formats delivered over lossy channels. The format provides the following set of tools:

— Reception data, which provides means to store the received data and document transmission information such as received or lost byte ranges and whether the corrupted/lost bytes are present in the file.

— Repair information, such as location of the source file, possible byte offsets in that source, byte stream position at which a parser can try processing a corrupted file; depending on the communication channel, this information may be setup by the receiver or through out-of-band means.

— File format specific information, which depends on the type of file stored as a partial file; this document only defines additional tools for files based on ISO/IEC 14496-12.

4.2.2 Data layout

The partial file format is designed to allow continuous recording and storing of received data in a flexible way. A partial file is composed of a header providing information valid for the entire recording followed by any number of segments; all of these segments are described by partial segment structures and are hence called partial segments, although some can be correctly received. The header does not contain any data from the source file and contains only information that can usually be inferred from the parameters of the transmission, and hence does not require editing of the header values upon finalizing the recording. Each partial segment contains a variable number of received bytes from a given source, a map of valid byte ranges for these received bytes and optionally some repair information. Byte maps give offsets relative to the first byte of data in this partial segment, rather than from the first byte of data in the source file, in order to process partial segments without any knowledge of their position in the partial file. Each partial segment may describe correctly received data, lost data or a mix of correctly received and loss data.

4.2.3 Box order

An overall view of the normal encapsulation structure is provided in Table 1. In the event of a conflict between this table and the prose, the prose prevails. The order of boxes within their container is not necessarily indicated in the table.

The table shows those boxes that may occur at the top-level in the left-most column; indentation is used to show possible containment. Not all boxes need to be used in all files; the mandatory boxes are marked with an asterisk (*). See the description of the individual boxes for a discussion of assumptions if the optional boxes are not present.

Objects using an extended type may be placed in a wide variety of containers, not just the top level.

The following rules shall be followed for the order of boxes in a partial file:

— The `FileTypeBox` shall occur first in the file.

— The `PartialFileBox`, if present, shall occur immediately after the `FileTypeBox`.

Table 1 — Box types, structure and cross-reference

	Box types, structure and cross-reference						
ftyp					*		*file type and compatibility*
pfil						5.1.1	*container for metadata global to file*
	pfhd				*	5.1.2	*data about the entire partial file*
	surl					5.1.8	*source URL and mime for the complete source file*
	ptle					5.1.7	*entry points information for the complete source file*
	fidx					5.2.2	*box file index for the complete source file*
		bidx				5.2.3	*box index for a box of the source file*
			dint			5.2.4	*data integrity container for the complete source file*
				dihd	*	5.2.5	*data integrity hash*
			frpa			5.2.6	*front part (initial bytes) of an indexed box*
	dref					5.1.1	*data reference box as defined in ISO/IEC 14496-12*
pdat						5.1.2	*partial segment data*
pseg					*	5.1.4	*partial segment*

Key

* Boxes that are mandatory within their container, which itself can be mandatory or optional.

Table 1 *(continued)*

	Box types, structure and cross-reference						
	pshd				*	5.1.5	*partial segment header containing information related to the partial segment*
	ploc				*	5.1.6	*partial segment location*
	surl					5.1.8	*source URL and mime of the source file for the partial segment*
	ptle					5.1.7	*entry point information of the source file for the partial segment*
	fidx					5.2.2	*box file index of the source file for the partial segment*
		bidx				5.2.3	*box index for a box of the source file for the partial segment*
			dint			5.2.4	*data integrity container of the source file for the partial segment*
				dihd	*	5.2.5	*data integrity hash*
			frpa			5.2.6	*front part (initial bytes) of an indexed box*

Key

* Boxes that are mandatory within their container, which itself can be mandatory or optional.

4.2.4 Mixed partial files

When the source file uses a format derived from ISO/IEC 14496-12, it can be desirable to encapsulate only the corrupted top-level boxes of the source file in a partial segment, while leaving the correctly received top-level boxes untouched. Figure 1 shows the case where certain movie segments have been received complete, other parts are only partial and there are yet other parts for which it is known that no information has been received. The incomplete parts map to one or several partial segments.

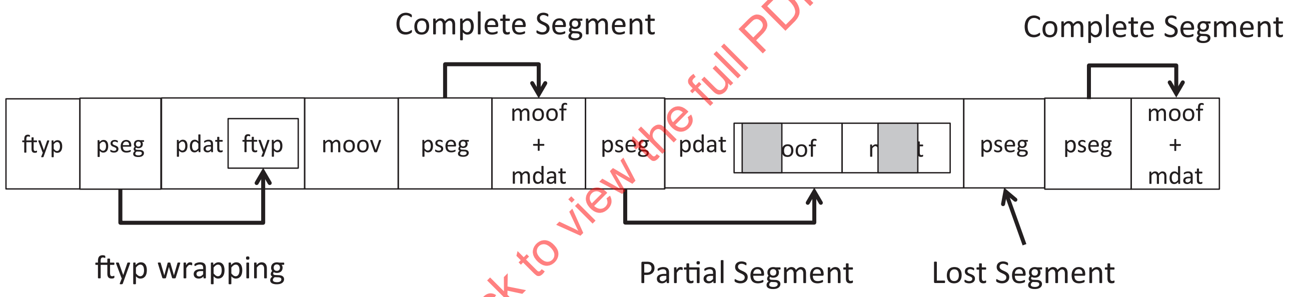

Figure 1 — Mixed partial file

This design is referred to as mixed partial file. In this configuration, the `FileTypeBox` shall contain a compatible brand of type `'pmff'`. The `FileTypeBox` of the source file, whether or not correctly received, shall not be modified; it shall be encapsulated in one partial segment and stored in a `PartialDataBox`, in order to ensure unicity of the `FileTypeBox` at the container level. Mixed partial file writers may insert, in the `FileTypeBox` of the partial file, brands from the source ISOBMFF file if the requirements of these brands are still met in the mixed partial file. This design allows backward compatibility with existing ISOBMFF file reader, as they would skip unrecognized boxes but still play the rest of the presentation.

The mixed partial file format is designed to store partially received fragments of a given ISOBMFF file, without any inspection or modification of the internal box structure of the source file. This format cannot be used to indicate, through in-file structures such as `SampleTableBox` or `TrackFragmentBox`, which items or samples are missing or corrupted. Such operation requires reprocessing of the source file and is possible using dedicated structures defined in ISOBMFF.

NOTE 1 In case an ISOBMFF segment made of multiple movie fragments is corrupted and the initial `SegmentTypeBox` is lost, a mixed partial file reader might have to insert a `SegmentTypeBox` at the appropriate place for correct processing by higher application layers.

NOTE 2 When ISOBMFF segments are stored in a mixed partial file, track timing can be lost if correctly received ISOBMFF segments do not contain `TrackFragmentBaseMediaDecodeTimeBox`.

4.2.5 Processing model

The processing model of partial files or mixed partial files is illustrated in Figure 2. In partial file mode, there is a “partial file handler” that is used, which in turn delegates to the original file handler. In mixed partial mode, the original file handler can be extended to handle partial files with their own original MIME type.

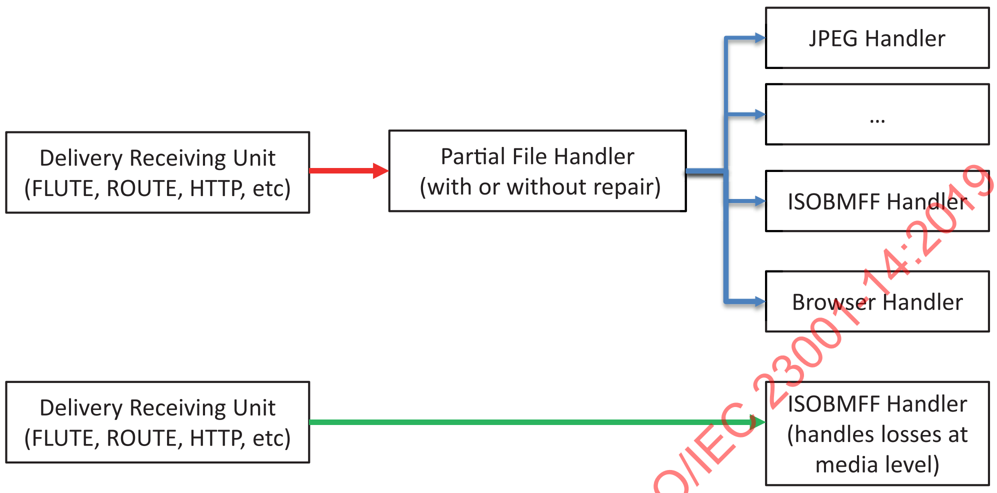
JPEG Handler
...
Delivery Receiving Unit (FLUTE, ROUTE, HTTP, etc)
Partial File Handler (with or without repair)
ISOBMFF Handler
Browser Handler
Delivery Receiving Unit (FLUTE, ROUTE, HTTP, etc)
ISOBMFF Handler (handles losses at media level)

Figure 2 — Processing model of partial files or mixed partial files

The source file shall be identical to the ordered concatenation of the data of each partial segment for that given source if no losses occurred or if all corrupted blocks were successfully repaired. A mixed partial file handler may typically optimize this process by not reconstructing the complete source file.

5 Box structures

5.1 Partial file general boxes

5.1.1 Partial file box

5.1.1.1 Definition

Box Type: 'pfil'

Container: File

Mandatory: No

Quantity: Zero or One

This box contains metadata information for the partial file. This metadata can be overridden with each partial segment.

A PartialFileBox may contain zero or one DataReferenceBox as defined in ISO/IEC 14496-12. Data references shall only be used to indicate an external location of (part of) the stored data; consequently, entries of the DataReferenceBox shall not use the flag value 0x000001.

NOTE Usage of data references allows building a partial file describing the recorded result of a transmission session stored in another format than the partial file format; for example, a partial file can be computed to describe the current reception state of a torrent session.

When enclosed, the BoxFileIndexBox provides a summary of the box hierarchy of the complete source file. It may be used to describe only type and size of a file-level box, or the complete or partial hierarchy of child boxes of one or several file-level boxes. Each top-level box of the source file will typically have zero or one associated BoxIndexBoxes in the BoxFileIndexBox.

If the set of boxes in the BoxFileIndexBox is also complete, the corresponding bit in file_flags should also be set.

If one BoxFileIndexBox is present in the PartialFileBox, it indexes (part of) the complete file and no other BoxFileIndexBox shall be present in any subsequent PartialSegmentBox.

5.1.1.2 Syntax

```
aligned(8) class PartialFileBox extends Box('pfil') {
   Box other_boxes[]
}
```

5.1.2 Partial file header box

5.1.2.1 Definition

Box Type: 'pfhd'

Container: PartialFileBox

Mandatory: Yes, if PartialFileBox is present

Quantity: Exactly one

This box contains general information about the partial file data and structures.

The following flag is defined for the PartialFileHeaderBox:

— complete_index: flag value is 0x000001; when set, indicates that the BoxFileIndexBox contained in this box is complete (the contents are always required to be dense and start at the beginning, so this means that the box documents all bytes to the end of the original box-structured file). This flag shall not be set for non-box-structured original files.

5.1.2.2 Syntax

```
aligned(8) class PartialFileHeaderBox extends FullBox('pfhd', 0,
flags) {

}
```

5.1.3 Partial segment box

5.1.3.1 Definition

Box Type: 'pseg'

Container: File

Mandatory: Yes

Quantity: One or more

The PartialSegmentBox is a container for metadata describing one or more blocks of received data from a given source, missing or corrupted bytes, source or indexing information. There are usually several instances of PartialSegmentBox in a partial file.

A PartialSegmentBox only contains boxes describing the source data; it does not contain any source data. Each PartialSegmentBox shall contain one PartialSegmentHeaderBox containing configuration of the partial segment; the associated source data may be located after or before the PartialSegmentBox, as described by the PartialSegmentLocationBox, and may be encapsulated in one or more PartialDataBox.

The resulting partial segment (i.e. the PartialSegmentBox and its associated data) shall conform to the object structure defined in subclause 4.1.

Since reconstruction of the source file is based on information from partial segment, storing a segment received without errors still requires the presence of a partial segment box describing the received data.

When enclosed, a BoxFileIndexBox provides a summary of the box hierarchy of the source file data in the current partial segment. It may be used to describe only type and size of a file-level box, or the complete or partial hierarchy of child boxes of one or several file-level boxes. Each top-level box of the source file will typically have zero or one associated BoxIndexBoxes in the BoxFileIndexBox.

If the set of boxes in the BoxFileIndexBox box is also complete, the corresponding bit in segment_flags should also be set.

If one BoxFileIndexBox is present in the PartialFileBox, it indexes (part of) the complete file and no other BoxFileIndexBox shall be present in any subsequent PartialSegmentBox.

5.1.3.2 Syntax

```
aligned(8) class PartialSegmentBox extends Box('pseg') {
   Box other_boxes[]
}
```

5.1.4 Partial segment header box

5.1.4.1 Definition

Box Type: 'pshd'

Container: PartialSegmentBox

Mandatory: Yes

Quantity: One per PartialSegmentBox

The PartialSegmentHeaderBox gives general information for the partial segment. The PartialSegmentHeaderBox shall be placed first in the PartialSegmentBox.

The following flags are defined for the PartialSegmentHeaderBox:

— last_segment: flag value is 0x000001. Presence of this flag indicates that this partial segment is the last one from the associated source; for a given file location as indicated by SourceURLBox, there should not be more than one partial segment with last_segment flag set in the PartialSegmentHeaderBox, and that partial segment should be the last one in the file. Readers may stop processing the file data from this source once this flag is encountered.

— no_further_repair: flag value is 0x00000002. This flag indicates that the repair operation(s) repaired the maximum information (no further repair would be beneficial), yet there may still be corrupted chunks in the content.

— complete_index: flag value is 0x000004; when set, has the same semantics as the complete_index flag defined for the PartialFileHeaderBox.

NOTE The `last_segment` flag can be used to detect end of file reception by file readers processing a file being recorded by another entity. The `no_further_repair` flag can be used to avoid repair attempts on content no longer available.

5.1.4.2 Syntax

```
aligned(8) class PartialSegmentHeaderBox extends FullBox('pshd',
version, flags) {
   if (version==0) {
      unsigned int(32) source_byte_offset;
   } else {
      unsigned int(64) source_byte_offset;
   }
   unsigned int(64) last_repair_time;
}
```

5.1.4.3 Semantics

`source_byte_offset` gives the offset in bytes in the associated file indicated by `SourceURLBox` of the first byte of the first chunk in this partial segment; value 0 means that this first byte of the first chunk is the first byte of the associated source file. The meaning of this field is unspecified if no `SourceURLBox` is present in the partial file.

`last_repair_time` gives the 64-bit NTP timestamp of the last repair process. The value shall be formatted in accordance with IETF RFC 5905. If 0, that time is unknown. File readers may use this information to decide whether attempting a repair on the file is useful or not (for example in scenario of file redistribution in a home network where the access point would have attempted the repair).

5.1.5 Partial data box

5.1.5.1 Definition

Box Type: `'pdat'`

Container: `File`

Mandatory: No

Quantity: Zero or more per `PartialSegmentBox`

The `PartialDataBox` contains source file data for this partial segment. In some cases, it can be desirable to have more data in this box than a 32-bit size would permit. In this case, the large variant of the size field, as defined in ISO/IEC 14496-12, can be used.

File readers should not assume file data contained in `PartialDataBox` can be safely inspected. For example, if the source file is an ISOBMFF file, it is likely that the structure of the file (such as boxes and samples) is corrupted. There is also no guarantee that reception blocks contained in `PartialDataBox` are stored in reception order.

5.1.5.2 Syntax

```
aligned(8) class PartialDataBox extends Box('pdat') {
   bit(8) data[];
}
```

5.1.5.3 Semantics

`data` contains source file data. The exact layout of the received bytes is given by the `PartialSegmentLocationBox`.

5.1.6 Partial segment location box

5.1.6.1 Definition

Box Type: 'ploc'

Container: PartialSegmentBox

Mandatory: Yes

Quantity: Exactly one per PartialSegmentBox

The PartialSegmentLocationBox contains a map of received and missed byte ranges from the source file for this partial segment, expressed as data chunks. Each chunk describes a lost or received byte range and its storage location in the partial segment.

When the list of chunks is not empty, it shall describe the complete, ordered set of bytes in the source file required to repair the partial segment.

A repair byte range for a corrupted or lost chunk may be built by summing the size of all previous chunks in the partial segment and adding the source_byte_offset indicated in the PartialSegmentHeaderBox.

The chunk data location is indicated by means of relative offset in the container file. For source files not based on the box structure defined in ISO/IEC 14496-12, the chunk data shall be stored in a PartialDataBox. For mixed partial files, chunk data may also be stored directly within the container file, provided that the resulting file is still compliant to both this document and ISO/IEC 14496-12; this implies that only completely received top-level boxes from the source file can be stored outside of PartialDataBox. Furthermore, when the source file derives from ISO/IEC 14496-12 and uses segments as defined in ISO/IEC 14496-12, a segment shall only be stored at partial file level if it is completely received without errors; otherwise, the partially received segment shall be stored in a PartialDataBox.

5.1.6.2 Syntax

```
aligned(8) class PartialSegmentLocationBox extends FullBox('ploc',
version=0, flags=0) {
   unsigned int(4)     length_size;
   unsigned int(4)     offset_size;
   unsigned int(16)    data_reference_index;
   unsigned int(16)    chunk_count;
   for (i=0; i < chunk_count; i++) {
      bit(1)    corrupted_chunk;
      bit(1)    data_present;
      bit(6)    reserved=0;
      unsigned int(length_size*8) chunk_size;
      if (data_present==1) {
         if (data_reference_index == 0) {
            signed int(offset_size*8) chunk_offset;
         } else {
            unsigned int(offset_size*8) chunk_offset;
         }
      }
   }
}
```

5.1.6.3 Semantics

length_size indicates the number of bytes used to encode chunk sizes in this box.

offset_size indicates the number of bytes used to encode chunk offsets in this box.

data_reference_index indicates the index of the data reference entry to use to retrieve data associated with the chunks. Data reference entries are stored in the DataReferenceBox in the PartialFileBox. A value of 0 indicates that the data is located in the containing file. A non-zero value indicates the index of the data reference entry, ranging from 1 to the number of data reference entries.

chunk_count indicates the number of chunks composing this partial segment. A value of 0 indicates that the partial segment is composed of a single chunk of unknown size; this may be used for example in mixed partial file to indicate the loss of an ISOBMFF segment without knowing its original size.

corrupted_chunk indicates that the chunk is corrupted. When corrupted_chunk is set, this indicates that this chunk data is corrupted or lost. When corrupted_chunk is not set, this indicates that this chunk data is valid data that has been received or repaired.

data_present indicates that the received data is present in the indicated file. When both data_present and corrupted_chunk are set, this indicates that all corrupted bytes have been kept, and missing bytes have been replaced with padding bytes; whenever padding is used, it is recommended to use 0x00 as the padded bytes value. When data_present is not set and corrupted_chunk is set, this indicates a missed or corrupted byte range, not stored in the partial segment.

chunk_size specifies the size of this chunk in bytes.

chunk_offset specifies the start offset of this chunk in the partial file. When the data reference index is 0 (data contained with the file), the offset is given as a signed number counting the number of bytes between the start of the containing PartialSegmentBox and the start of the chunk; a negative offset implies that the chunk data is located before the PartialSegmentBox. When the data reference is not 0 (data contained in another file), the offset is given as a positive number counting the number of bytes between the start of the chunk and the start of the file.

NOTE During the repair process, corrupted_chunk is set to 0 only after the chunk has been successfully repaired. When data_present is set, repair can be done by replacing the padding data with valid data. When data_present is not set, valid data is added during the repair process, data_present is set to 1 and chunk_offset of this chunk is added after the repair process completion. In this case, the chunk_offset of other chunks might need to be modified.

5.1.7 Partial top level entry point box

5.1.7.1 Definition

Box Type: 'ptle'

Container: PartialSegmentBox or PartialFileBox

Mandatory: No

Quantity: At most one per PartialSegmentBox, or one in PartialFileBox

The PartialTopLevelEntryPointBox indicates one or several top-level (file level) offsets in the source file at which a file reader may resume parsing in case of corrupted data. This allows a file reader to resynchronize file structure parsing in case of corrupted data impacting one of the previous top-level structures.

NOTE 1 For example, if the source file is a fragmented ISOBMFF file, the offsets can be used to indicate the position of different MovieFragmentBox in the file, allowing a parser to skip a series of fragments with errors, possibly removing fragments with errors.

NOTE 2 This information is usually transported out-of-band or through well-protected packets with more FEC in the transport layer; for example, the information can be described in the FDT of the file in a FLUTE session.

If this box is present in the PartialFileBox, it shall indicate the entry points for the complete file using absolute offsets, and no other PartialTopLevelEntryPointBox shall be present in any subsequent PartialSegmentBox.

The following flag is defined for the PartialTopLevelEntryPointBox:

— relative_offset: flag value is 0x000001. Presence of this flag indicates that indicated offsets are relative to the first byte of the first chunk of the partial segment containing this box. Absence of this flag indicates that indicated offsets are relative to the beginning (first byte) of the source file. This flag shall not be set if the container box is a PartialFileBox.

5.1.7.2 Syntax

```
aligned(8) class PartialTopLevelEntryPointBox extends
FullBox('ptle', version, flags) {
   unsigned int(32)   entry_count;
   for (i=0; i < entry_count; i++) {
      if (version==1) {
         unsigned int(64)   index_offset;
      } else {
         unsigned int(32)   index_offset;
      }
   }
}
```

5.1.7.3 Semantics

entry_count is the number of entry points listed in this box.

index_offset specifies the offset of the entry point in the source file. If version 1 is used, 64-bit data offsets are used; otherwise 32-bit data offsets are used.

5.1.8 Source URL box

5.1.8.1 Definition

Box Type: 'surl'

Container: PartialSegmentBox or PartialFileBox

Mandatory: No

Quantity: Zero or more

The SourceURLBox is used to indicate a source URL and associated MIME type of the source file. It is typically inserted in PartialFileBox or PartialSegmentBox by the receiver and can be used by a file reader to repair the file.

There may be several SourceURLBox in a partial file. SourceURLBox declared in PartialFileBox define sources valid for the entire partial file; SourceURLBox declared in PartialSegmentBox define sources valid for the partial segment only, in which case SourceURLBox declared in PartialFileBox shall be ignored. If no SourceURLBox is declared in a PartialSegmentBox, the SourceURLBox declared in PartialFileBox, if any, can be used to identify source files. When several instances of SourceURLBox are declared for a partial segment, all URLs given in these SourceURLBox are equally valid for a repair process.

NOTE 1 Using several different SourceURLBox allows for gathering in a single file several files transmitted over the network, such as a layered coded media where each layer is carried in a separate file.

NOTE 2 A partial file writer can avoid inserting a SourceURLBox in each segment by inserting a SourceURLBox in PartialFileBox.

5.1.8.2 Syntax

```
aligned(8) class SourceURLBox extends FullBox('surl', 0, 0) {
   string url;
   string mime;
}
```

5.1.8.3 Semantics

`url` is a NULL-terminated C string encoded in UTF-8; the last NULL character shall be set even if the URL is empty. The URL specifies a source URL for the source file data in the partial file or partial segment. This may identify the physical network or an alternative URL where the file can be found for later repair.

`mime` is a NULL-terminated C string encoded in UTF-8; the last NULL character shall be set even if the mime is empty. It specifies the mime type associated with the file at the given URL.

5.2 Partial file ISO base media file helpers

5.2.1 General

The boxes defined in this subclause shall only be used when the source file is following a specification derived from the ISO/IEC 14496-12. These boxes use the box-based structure information of the source file to provide further details for parsing or validating of a partial file or partial segment.

5.2.2 Box file index box

5.2.2.1 Definition

Box Types:	`'fidx'`
Container:	`PartialSegmentBox` or `PartialFileBox`
Mandatory:	No
Quantity:	Zero or one within the `PartialFileBox`. Either exactly zero per `PartialSegmentBox`, if the `PartialFileBox` contains `BoxFileIndexBox`; or at most one per `PartialSegmentBox` otherwise.

The `BoxFileIndexBox` provides a summary of the box hierarchy of a box-structured file. It contains a set of `BoxIndexBox` boxes, each of which describes one top-level box. Each top-level box of the source file will typically have zero or one associated `BoxIndexBoxes` in the `BoxFileIndexBox`.

NOTE This information is usually transported out-of-band or through well-protected packets in the transport layer; for example, the information can be described in the FDT of the file in a FLUTE session.

The set of bytes documented by the set of contained `BoxIndexBox` boxes shall start with the first byte in the associated source file if the `BoxFileIndexBox` is present in a `PartialFileBox`, or with the byte starting at `source_byte_offset` in the associated source file if the `BoxFileIndexBox` is present in a `PartialSegmentBox`. This shall apply regardless of the location of the `BoxFileIndexBox` (embedded in the indexed file or stored outside the file).

The set of documented boxes shall be both in-order (in the same order as the boxes they are indexing in the source file) and “dense” i.e. document a contiguous set of original bytes. The `complete_index` flag in the `PartialFileHeaderBox` or external signalling can indicate whether the set is also complete.

5.2.2.2 Syntax

```
aligned(8) class BoxFileIndexBox extends Box('fidx') {
   BoxIndexBox   other_boxes[];         // to end of the box
}
```

5.2.3 Box index box

5.2.3.1 Definition

Box Types 'bidx'

Container: BoxFileIndexBox or BoxIndexBox

Mandatory: No

Quantity: Zero or More

The BoxIndexBox provides a summary of the box hierarchy of a box.

If the described box does contain child boxes, the other_boxes array may contain zero or more instances of BoxIndexBox. BoxIndexBox boxes are not required, i.e. the depth of the nesting documented is at the choice of the writer. If contained BoxIndexBox boxes occur, however, they shall be complete and document all bytes of the enclosing original box, in order.

If the described box does not contain any child boxes, the other_boxes array shall not contain any BoxIndexBox.

If the described box has some fields that are not deriving from boxes, the other_boxes array may contain zero or one FrontPartBox. Otherwise, the other_boxes array shall not contain any FrontPartBox.

NOTE The contents of this box follow exactly the syntax for a Box in ISO/IEC 14496-12.

5.2.3.2 Syntax

```
aligned(8) class BoxIndexBox extends Box('bidx') {
   unsigned int(32) indexed_box_size;
   unsigned int(32) indexed_box_type;
   if (indexed_box_size==1) {
      unsigned int(64) indexed_box_largesize;
   } else if (indexed_box_size ==0) {
      // original box extends to end of original file
   }
   if (indexed_box_type=='uuid') {
      unsigned int(8)[16] indexed_box_usertype;
   }
   Box   other_boxes[];         // to end of the box
}
```

5.2.3.3 Semantics

indexed_box_type is the type of the indexed box in the described file.

indexed_box_size is the 32-bit size of the indexed box in the described file. If the indicated size is 0, this means that the indexed box extends until the end of the indexed container, as specified in ISO/IEC 14496-12.

indexed_box_largesize is the 64-bit size of the indexed box in the source file.

indexed_box_usertype is the extended type of the indexed box in the described file when user extension is used.

other_boxes is a list of boxes containing BoxIndexBox, DataIntegrityBox or FrontPartBox.

5.2.4 Data integrity box

5.2.4.1 Definition

Box Type: 'dint'

Container: BoxFileIndexBox or BoxIndexBox

Mandatory: No

Quantity: Zero or one per BoxIndexBox

The DataIntegrityBox contains integrity data information for a set of boxes. A DataIntegrityBox always refers to the indexed box described by its parent BoxIndexBox.

5.2.4.2 Syntax

```
aligned(8) class DataIntegrityBox() extends Box('dint') {
}
```

5.2.5 Data integrity hash box

5.2.5.1 Definition

Box Type: 'dihd'

Container: DataIntegrityBox

Mandatory: Yes if DataIntegrityBox is present

Quantity: One or more per DataIntegrityBox

The DataIntegrityHashBox carries cryptographic hash information for a set of boxes or samples. This includes the algorithm used for computing the cryptographic hash, the cryptographic hash itself and which and how boxes are used to produce the cryptographic hash.

Three flavors of the data integrity protection are specified, discriminated using the mode field of the DataIntegrityHashBox.

— When a DataIntegrityBox contains a DataIntegrityHashBox with mode set to zero, it indicates to the reader that the integrity is computed for the indexed box described by the DataIntegrityBox, i.e. for the entire content indexed by the BoxFileIndexBox or the BoxIndexBox that contains the DataIntegrityBox.

— There shall be a maximum of one DataIntegrityHashBox with mode set to zero in the parent DataIntegrityBox.

— When a DataIntegrityBox contains a DataIntegrityHashBox with mode set to one, it indicates to the reader that the integrity is computed for child boxes of the indexed box described by the DataIntegrityBox.

— There can be multiple DataIntegrityHashBox with mode set to one in the parent DataIntegrityBox.

— When a DataIntegrityBox box contains a DataIntegrityHashBox with mode set to two, it indicates to the reader that

  — the sub_mode in the DataIntegrityHashBox box is one of 'stsz', 'trak' or 'sgpd', and

  — the hash is computed over specific types of sample information, described by the DataIntegrityHashBox itself.

  There shall be at most one DataIntegrityHashBox with mode set to two and sub_mode set to 'stsz' or 'trak' in the parent DataIntegrityBox. There may be multiple instances of DataIntegrityHashBox with mode set to two and sub_mode set to 'sgpd' in the parent DataIntegrityBox; this type of DataIntegrityHashBox is used to provide the sanity check information of coded samples associated with a given sample group.

5.2.5.2 Syntax

```
aligned(8) class DataIntegrityHashBox () extends FullBox('dihd', 0,
flags=0) {
   unsigned int(16) mode;
   unsigned int(16) algorithm;

   if (mode==0)
      num_hashes = 1;
   else if (mode==1) {
      unsigned int(32) num_hashes;
   } else if (mode==2) {
      unsigned int(32) sub_mode;
      if (sub_mode == 'sgpd') {
         unsigned int(32) num_hashes;
      }
   }
   for (i=0; i< num_hashes; i++) {
      if (mode == 1) {
         unsigned int(32) box_4cc;
      } else if ((mode == 2) && (sub_mode == 'sgpd')) {
         unsigned int(32) grouping_type;
         unsigned int(32) num_entries;
         for(i=0; i<num_entries; i++} {
            unsigned int(32) group_description_index[i];
         }
      }
      bit(nb_hash_bits) hash_value;
   }
}
```

5.2.5.3 Syntax

mode indicates the type of integrity check used. The valid values of this field are 0, 1 and 2. Other values are reserved. When the mode field is set to 0, the hash string is computed over the concatenated bytes of the indexed box described by this box. When the mode field is set to 1, the hash is computed over the concatenated bytes of child boxes contained in the indexed box described by this box. When the mode field is set to 2, the type of integrity check is indicated by the sub_mode field. A DataIntegrityBox with DataIntegrityHeaderBox with mode set to two can be contained only in a BoxIndexBox indexing SampleTableBox or TrackFragmentBox.

algorithm indicates the algorithm used for computing the cryptographic hash. The algorithms are specified using four character codes, as follows:

Algorithm 4CC	Algorithm used
'md5 '	MD5 checksum as specified in RFC 6151
'sha1'	SHA-1 checksum as specified in RFC 3174

num_hashes indicates the number of hashes present in the box. When not present, the value 1 is inferred.

sub_mode indicates, using a four character code, the samples over which the cryptographic hash is computed. The following four character code are specified:

— 'stsz': the hash string is computed over the concatenated 32-bit sizes of the samples in the track or track fragment.

— 'trak': the hash string is computed over the concatenated bytes of samples that are in the track or track fragment.

— 'sgpd': the hash string is computed over concatenated bytes of samples that are mapped to selected entries of a group of the indicated type.

box_4cc indicates the four character code of the child boxes of the indexed box described by this box, used for computing the hash. If this value is 0, all child boxes of the indexed box described by this box are used for computing the hash.

nb_hash_bits indicates the number of bits used to code the hash string. When the algorithm field value is 'md5 ', this field takes the value 128. When the algorithm field value is 'sha1', this field takes the value 160.

hash_value indicates the resulting cryptographic hash for the given mode and parameters.

grouping_type indicates the sample group type used when sample groups are used for the hash calculation; only samples associated to this grouping type will be considered samples for the hash string calculations.

num_entries indicates the number of sample group entries to consider when selecting samples for cryptographic hashing; if 0, all sample mapped to any entry of the sample group of the given grouping type are considered for cryptographic hash; otherwise, the cryptographic hash is derived from those samples mapped to the indicated sample group description indices.

group_description_index[i] specifies that the checksum is derived from samples mapped to the sample group description index equal to group_description_index[i].

5.2.6 Front part box

5.2.6.1 Definition

Box Types 'frpa'

Container: BoxIndexBox

Mandatory: No

Quantity: Zero or One

The FrontPartBox provides a selected number of initial bytes of the content of the box identified by the containing BoxIndexBox. The number of bytes carried should be “small” and may be limited by the application, profile or specification (e.g. 256 bytes). The content of the box starts with the first byte following the fields defined for the class Box in ISO/IEC 14496-12.

NOTE This information can be used for carrying metadata that is essential for interpreting the file structure within the file index.

5.2.6.2 Syntax

```
aligned(8) class FrontPartBox extends Box('frpa') {
   unsigned int(8)   box_content[];
}
```