NGA.SIG.0045_0.5_ISOHB

2023-08-31

NGA STANDARDIZATION DOCUMENT

Standard Information/Guidance (SIG)

ISO Base Media File Format (ISOBMFF) Handbook for DoD/IC/NSG Applications

(2023-08-31)

Version 0.5

(DRAFT)

NATIONAL CENTER FOR GEOSPATIAL INTELLIGENCE STANDARDS

Contents

Table of Figures

Table of Tables

Introduction

The ISO Base Media File Format (ISOBMFF) Handbook for DoD/IC/NSG Applications provides introductary and guidance for government-developed imagery standards (e.g., GEOINT Imagery Media for ISR – GIMI) built upon the commercial ISOBMFF standard (ISO/IEC 14496-12 [1]) and its derived ecosystem of profiles. This ecosystem of standards supports the formatting, transmitting, receiving, and processing of video, audio, imagery, and imagery-related information.

This document is a Standard Information/Guidance (SIG) document. It introduces common concepts and principles for government-developed standards based on ISOBMFF and its profiles. As such this document does not introduce new requirements but assumes adherence to requirements within the referenced ISO standards.

Revision History

Version	Approval Date	Change Description
0.5	2023-08-31	Initial Version

1 Purpose

The ISOBMFF standard and its derivative standards form a collection of commercial standards supporting many different applications. The commercial community brings together various vendors which provide large investments to build, test, and deploy the ISOBMFF collection of standards. The ISOBMFF provides a solution for interoperability between encoders, distribution, and decoder/players. Companies are actively building both hardware and software to support the ISOBMFF standards.

NGA leverages this commercial investment by defining profiles/extensions of the ISOBMFF for developing standards specific to the needs of the National System for Geospatial Intelligence (NSG). The resulting profiles/extensions contribute to creating an integrated, efficient, effective, and interoperable GEOINT enterprise and supports the broader goals of modernizing and improving the GEOINT analytic workflow by:

- Developing a container solution for Still, MSI, HSI, Motion Imagery, and metadata for NSG applications
- Developing a long-term sustainment model (e.g., collaboration with industry) for the container solution
- Reducing costs by leveraging industry investments, technologies, tools, and solutions
- Improving the efficiency of operations by increasing timely access to information for users with cloud-friendly performance features
- Aids in automation and interoperability consistent with AI/ML developments
- Facilitating the effective search, discovery, and retrieval of information
- Enables community wide use of OSS/GOSS libraries and tools to aide adoption and transition of the new capability

This document, ISOBMFF Handbook for DoD/IC/NSG Applications, is a bridge between the commercial ISOBMFF ecosystem of standards and government-developed standards, such as NGA.STND.0076 – GIMI [2]. Leveraging ISOBMFF comes with a steep learning curve. This document serves to shorten this learning curve by summarizing the ISOBMFF components needed in developing profiles/extensions for government use. This document focuses on two industry format standards, ISOBMFF and HEIF (ISO/IEC 23008-12 [3]). These two documents are the basis of other industry standards such as, JPEG 2000 (ISO/IED 15444-1 [4]), AVC (ISO/IEC 14496-10 [5]), HEVC (ISO/IEC 23008-2 [6]), and the Uncompressed data standard (ISO/IEC 23001-17 [7]).

Figure 1 depicts the relationship of this document with respect to the ISOBMFF-based standards and NGA standards. The ISOBMFF and HEIF Format Standards (dark green) are the basis for many other Codec Standards such as JPEG 2000, AVC, HEVC, and the Uncompressed data standard (light green). At the top of the diagram, the Government Profiles (blue and grey) define augmentations and restrictions of the ISO documents to ensure interoperability of government data and resources. The ISOBMFF Handbook (red) summarizes the contents of the ISOBMFF standard. The HEIF standard will be a future update.

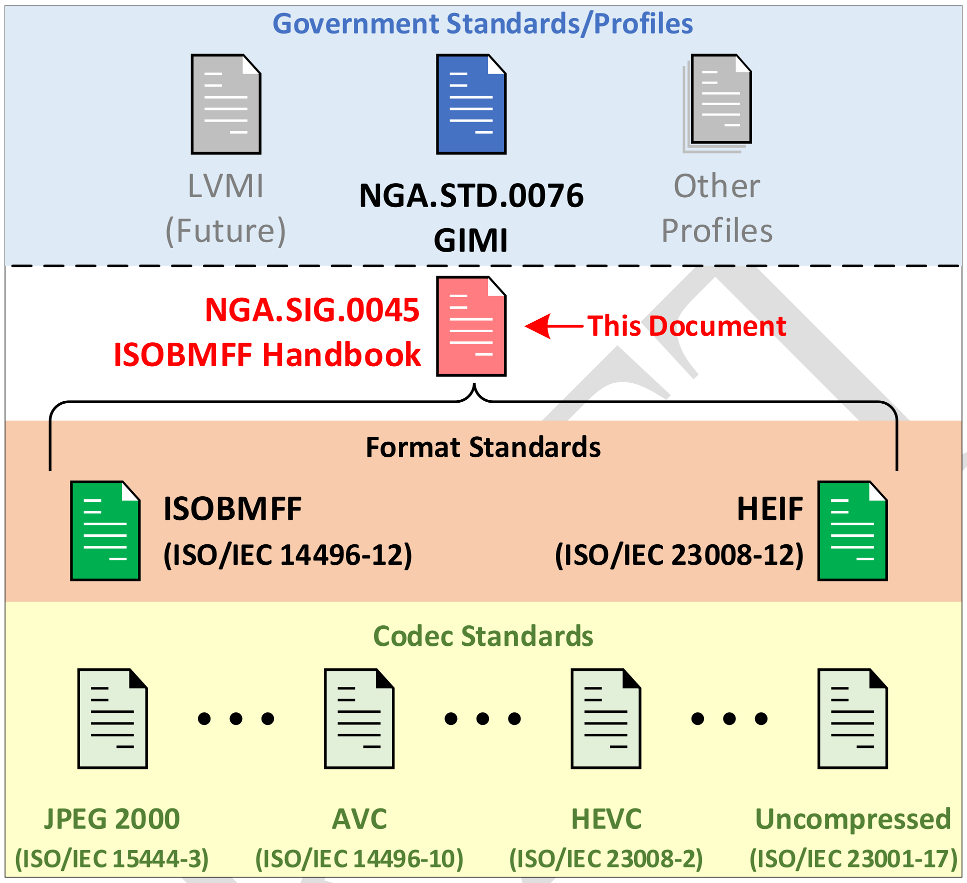


Figure 1: Relationship of ISOBMFF Standards, ISOBMFF Handbook, and NGA Profiles

2 References

[1] ISO/IEC 14496-12 Information technology - Coding of audio-visual objects - Part 12: ISO base media file format.

[2] NGA NGA.STND.0076.0.3 Geospatial-Intelligence (GEOINT) Imagery Media for Intelligence, Surveillance, and Reconnaissance (ISR), 31 08 2023.

[3] ISO/IEC 23008-12:2022 Information technology - High efficiency coding and media delivery in heterogeneous environments - Part 12: Image File Format, 2022.

[4] ISO/IEC 15444-1:2019 Information technology - JPEG 2000 image coding system: Part 1: Core coding system.

[5] ISO/IEC 14496-10:2020 Information Technology - Coding of audio-visual objects - Part 10: Advanced Video Coding.

[6] ISO/IEC 23008-2 Information technology - High efficiency coding and media delivery in heterogeneous environments - Part 2: High efficiency video coding.

[7] ISO/IEC 23001-17 Information technology — MPEG Systems technologies — Part17: Uncompressed video and images in ISO Base Media File Format.

[8] ITU-T X.667 | ISO/IEC 9834-8 Procedures for the generation of universally unique identifiers (UUIDs) and for their use in the international object identifier tree under the joint UUID arc, 14 Oct 2012.

3 Terms, Definitions and Acronyms

The purpose of providing terminology is to clarify the meaning and intent of words used in this document by giving the reader instruction and context on how they are to be interpreted, with an emphasis on terms that might otherwise be ambiguous.

3.1 Terms and Definitions

The document uses the following terms and definitions:

4CC	An ASCII encoded "four-character code" indicating data formatting requirements.
Box	A packet of data in an ISOBMFF file consisting of a header and payload. The header includes a size, type, and other information.
Brand	An identification of a set of specifications for use within and ISOBMFF file. Brands indicate requirements for producers when generating files and for readers when decoding, interpreting, and presenting content.
Composition Timeline	The display order of a set of samples in a track.
Decoding Timeline	The decode order of a set of samples in a track.
ISOBMFF-Core	A term used in this document to denote the foundational capabilities of ISOBMFF derived from boxes defined in the ISO/IEC 14496-12 standard.
ISOBMFF-Family	A suite of related standards all derived from ISO/IEC 14496-12 (ISOBMFF), such as HEIF, MP4, NAL file format, CMAF, OMAF, etc.
Presentation Timeline	The playout order of a set of samples from a track.
Sample	All the data associated with a single time.
Track	A set of samples and the characteristics for the samples.

3.2 Acronyms and Initialisms

This document uses the following acronyms:

4CC	Four-character code
ABR	Adaptive Bitrate Streaming
DT	Decode Times
EOF	End of File
GEOINT	Geospatial-Intelligence
GIMI	GEOINT Imagery Media for ISR
HEIF	High Efficiency Image File Format

IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
ISOBMFF	ISO Base Media File Format Family of Standards
ISOBMFF-Core	ISOBMFF (ISO/IEC) 14496-12
NSG	National System for Geospatial Intelligence
RLE	Run Length Encoding
SDL	Syntactic Description Language
UTF8	(Universal Coded Character Set) Transformation Format – 8-bit
UUID	Universally Unique Identifier

4 Overview

ISOBMFF refers to both a singular standard ISO/IEC 14496-12 and an ecosystem of standards that extend ISO/IEC 14496-12. This document distinguishes a pure implementation of ISO/IEC 14496-12 from the ecosystem implementations (e.g., HEIF) by using two terms: ISOBMFF-Core and ISOBMFF-Family. ISOBMFF-Core indicates definitions, rules, or implementations of ISO/IEC 14496-12. ISOBMFF-Family indicates the ecosystem of implementations based on the ISOBMFF-Core.

The ISOBMFF-Core standard defines a container file for video, metadata, and audio; media with a time-base. ISOBMFF-Family standards build on the ISOBMFF-Core to provide additional capabilities. For example, the HEIF standard builds upon the ISOBMFF-Core fundamentals to add support for Still Images. The ISOBMFF-Family of standards has broad adoption by industry resulting in an installed ecosystem of common tools and applications (see Section 4.1).

The ISOBMFF-Core and ISOBMFF-Family standards use the Syntactic Description Language (SDL) to describe content and its formatting. Understanding SDL is important to understand the structures, rules, and data parameters within all the ISOBMFF standards. SDL is an object-oriented underlying framework of "classes" which convert to a bitstream.

Section 5 provides background information about the ISOBMFF-Core standard. Section 5.1 is a high-level, non-technical overview about ISOBMFF-Core files. Section 5.2 defines the ISOBMFF-Core file structure consisting of "boxes" of information. This section introduces SDL details and box diagrams. Section 5.3 describes the ISOBMFF-Core Base Box Structure consisting of specific box types. This document defines the Base Box Structure as somewhat of a "minimum" set of boxes for an ISOBMFF-Core ("somewhat" because ISOBMFF-Core does not have rigid requirements for a minimum structure). Section 5.4, Section 5.5, and Section 5.6 provide specifics of the Base Box Structure.

Section 6 is a placeholder for the HEIF document.

4.1 Use of commercial libraries

All topics within this document have industry tools and libraries to create / read ISOBMFF files. The following lists a sample of available open-source tools:

ISOBMFF Tools	Purpose
GPAC MP4Box	On-line web based ISOBMFF Box Structure Viewer
Dash-Industry-Forum isoboxer	A lightweight browser-based MPEG-4 (ISOBMFF) file/box parser
FFmpeg ffprobe	Quickly gather details on media container internals
mp4viewer	Viewer in Python
Bento4	Full-featured MP4 format and MPEG DASH library and tools (in C++, with Java and Python bindings)
mp4parser	A Java library to read, write and create MP4 files
isoviewer	Java desktop application to inspect ISO 14496-12 and other MP4 files

5 ISO Base Media File Format

This section focuses on the construction of ISOBMFF-Core files.

5.1 High-Level Overview

The ISOBMFF-Core uses a presentation file concept for defining how to describe and store media (e.g., video and audio). Per ISOBMFF-Core, a presentation file contains the data which structures, orders, times, and describes the media data (video) player uses to present to users. A presentation file contains a single movie and administration information.

In viewing a movie, the user sees a continuous series of images, potentially with audio and textual information (e.g., closed captioning). The purpose of ISOBMFF-Core is to provide the source of the images, audio, and textual information needed for the presentation, along with the administration information necessary to display the presentation. ISOBMFF-Core does not limit the source data to only what a person sees and hears, but potentially includes a superset of imagery, audio, and other information; the administration information determines what subset of source data the person will see.

Figure 2 illustrates the presentation process. The bottom of the figure shows a single ISOBMFF-Core file (blue) which contains Timed Media (Video, Audio, Text/other) (yellow) and Administration (green) data. A Decoder (orange) and Player (red) use the information from the ISOBMFF-Core file to create a media sequence consisting of rendered audio, text (T), and images, which ultimately displays on a screen for viewing or other processing. In this document, a player orchestrates the flow of data using the administration data to aid in what, when, and how to build the media sequence. In this document, decoding is the process which converts individual timed media data types into playable imagery, audio, and text; a decoder performs decoding. In this document encoding is the process for converting raw data and organizing the data into an ISOBMFF-Core file; an encoder (not shown) performs encoding.

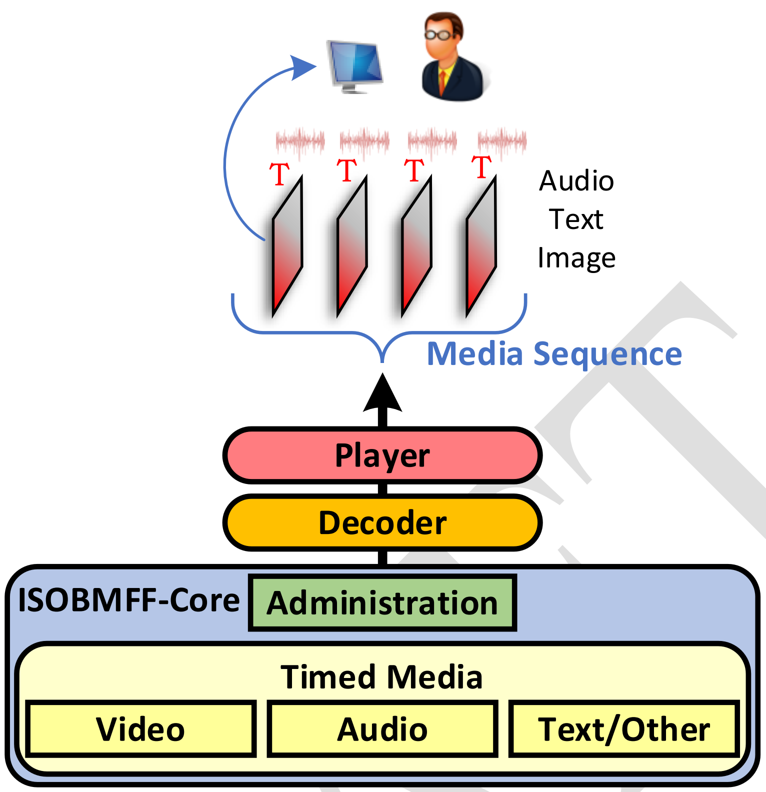


Figure 2: Presentation Process

From the computer science perspective, ISOBMFF-Core stores all its information in a “box” architecture. Boxes are the same as variable length data packets containing a box-length, box-type, and box-data. Each ISOBMFF-Core file is a series of boxes; Section 5.2 provides details of this perspective.

The ISOBMFF-Core standard provides a specific organization of the boxes essential to playing the “movie”.

5.1.1 ISOBMFF-Core Organization

While the ISOBMFF-Core contains a single movie, the movie is an aggregation of one or more separate parts. Movies contain different timed media types, such as video, audio, and timed text. The ISOBMFF-Core standard separates and organizes each of the timed media types independently yet provides the administrative information for them to seamlessly work together during playout. The standard furthermore divides each timed media type into a sequence of parts, called tracks, each of which contains its own administrative information.

Figure 3 illustrates a Movie (blue) consisting of three timed media types, Text (green), Audio (red), and Imagery (orange). Conceptionally, all three are one unit of “media”; however, ISOBMFF-Core breaks all the timed media into separate tracks. The illustration shows the Imagery divided into 4 tracks (Tracks$_1$-Track$_4$), the Audio divided into two tracks (Track$_5$ and Track$_6$), and the Text (Track$_7$) as one track. The figure illustrates the media does not have to be contiguous, i.e., there is a large gap in the imagery between tracks 2 and 3.

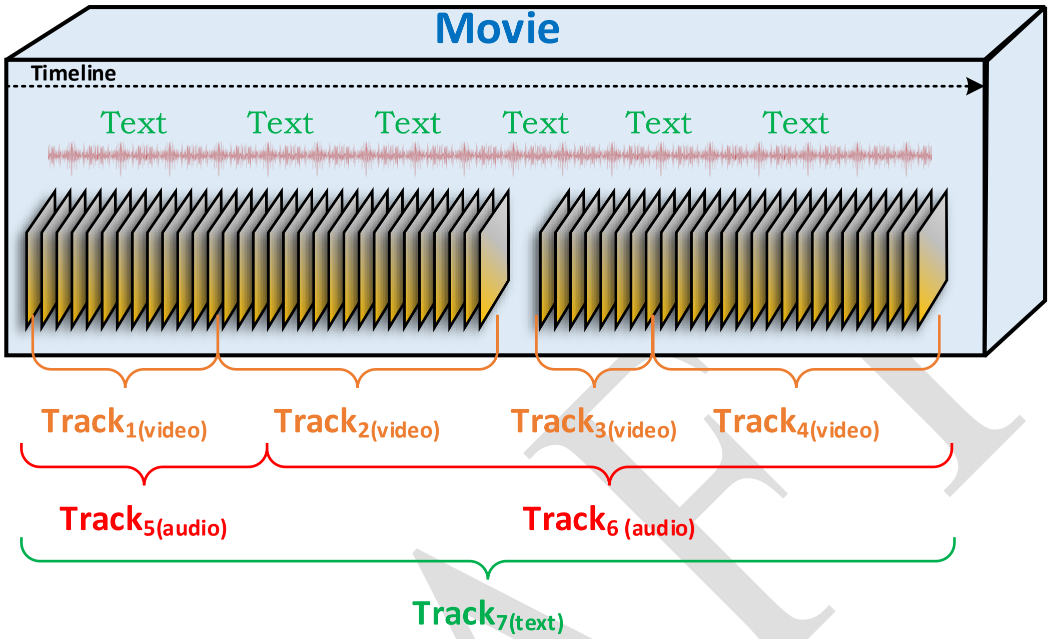


Figure 3: ISOBMFF-Core Movie

Encoders determine when to separate media into different tracks. There are many reasons why an encoder builds separate tracks including gaps in timeline or changes to: encoding methods, encoding rates, image sizes, image frame rates. Furthermore, movies may include tracks which are not a part of the intended playback but are available for inspection as needed.

ISOBMFF-Core further defines each track as a collection of media samples and administrative information. A sample is a unit of media, such as one image, one audio sample, etc. Each track's administrative information provides timing, ordering, and data storage location.

ISOBMFF-Core separates the raw imagery data from the administrative data into two different areas of the file. The imagery data is one or more large data blocks, typically at the end of the file. The administrative information provides all the information about the movie, tracks, timing, etc. including "pointers" to the actual imagery data, usually in the large data blocks at the end of the file. The administration information is typically in the beginning of the file in a movie structure hierarchy.

Figure 4 illustrates the typical high-level ISOBMFF-Core file organization as a hierarchy. The hierarchy consists of File Information (Section 5.1.2), a single Movie Structure (Section 5.1.3), and the Media Data (Section 5.1.4). The File Information is data about the contents of the file to aid in decoding and data interpretation. The Movie Structure includes Movie Information, and a series of Tracks (1…n). Each track contains Track Information, Track Associations, Edit List, and Media/Sample Details. The Media/Sample Details includes "pointing" to the data in Move Data. The Media Data is large blocks of media.

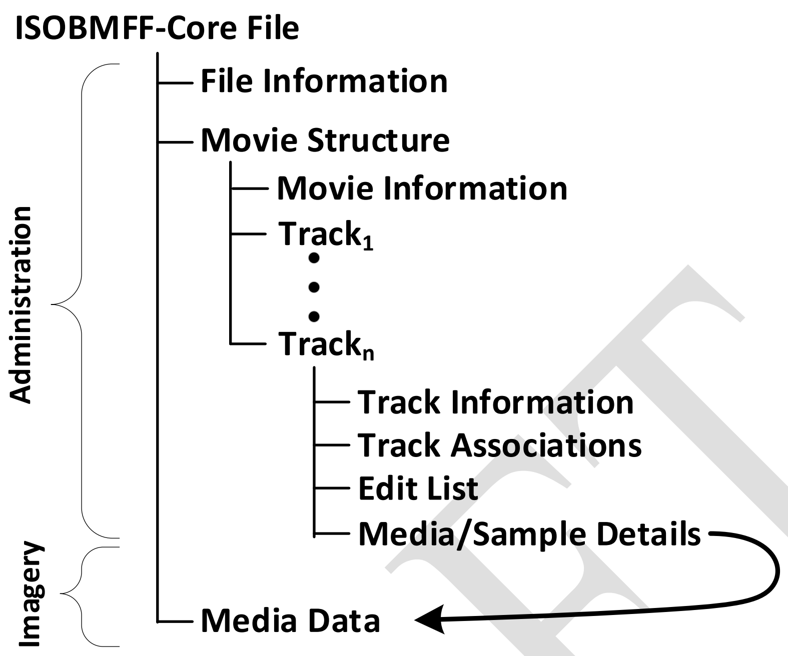


Figure 4: Presentation Organization

5.1.2 File Information

File Information aids in understanding and processing the whole file. The types of information include Brands and File-Level metadata.

Brands identify the specifications in use within the file; they indicate requirements for producers when generating files and for readers when decoding, interpreting, and presenting content.

See Section 5.4 for File Information details and boxes.

5.1.3 Movie Structure

ISOBMFF-Core provides two different Movie Structures, fragmented and non-fragmented. Fragmentation sub-divide tracks into smaller parts called fragments to support Adaptive Bitrate Streaming (ABR). This version of the ISOBMFF Handbook does not describe fragmentation.

The non-fragmented movie structure is a hierarchy of data consisting of Movie Information and a list of Tracks. The Movie Information provides wholistic administrative movie data, such as overall timing information, image size, encoder/decoder, information, etc.

The list of Tracks (Section 5.1.3.1) provides the administration data to playback track data as a presentation.

See Section 5.5 for Movie Structure details and boxes.

5.1.3.1 Tracks

All timed media in a movie is in the form of a track. Each type of timed media becomes its own track, i.e., video, audio, and text each become their own independent track. A stream of a media type can use one or more tracks, as necessary.

Each track contains Track Information, Track Associations, Edit List, and Media/Sample Details.

The Track Information includes track type, identifier, creation/modification times, duration, dimensions for imagery or text, audio volume, track dependency, user data, transformation matrix and other metadata.

Track Associations enable making track dependencies and groups. For example, audio and video tracks have timing dependencies, so one references the other. When multiple tracks share a common characteristic, encoders define a track group.

The Edit List is a set of instructions on how the player presents the track to the end user.

Media/Sample Details includes creation time, modification time, timescale, duration, language, media type, sample timing, sample ordering, sample location and other metadata.

See Section 5.5.1.1 for Track details and boxes.

5.1.4 Media Data

The Media Data is one or more large blocks of bytes to store the media. The Media Data is where the imagery, audio, and other data resides; however, it is useless without the Movie Structure administrative data to provide meaning to the different areas of the Media Data.

See Section 5.6 for Media Data Details and boxes.

5.2 ISOBMFF File and Box Structure

This section summarizes the Object-Structured File Organization section of ISOBMFF 14496-12.

5.2.1 File Structure

ISOBMFF-Core defines a binary file structure organized as a series of well-defined blocks of bytes, called boxes. Boxes are variable in length and have different content and purposes. Each box has its own size, B_N, where N represents the number of bytes in the box. Figure 5 illustrates an ISOBMFF-Core file containing a series of boxes, B_1 through B_M. Each box, i =1 to M, has its own size, $N_i = Size(B_i)$ with internal byte addressing from 0 to $N_i - 1$. The overall file has a size F_S, equal to the sum of all the box sizes in bytes, i.e., $F_S = \sum_{i=1}^{M} B(i)_N$, and with an addressing range from 0 to $F_S - 1$.

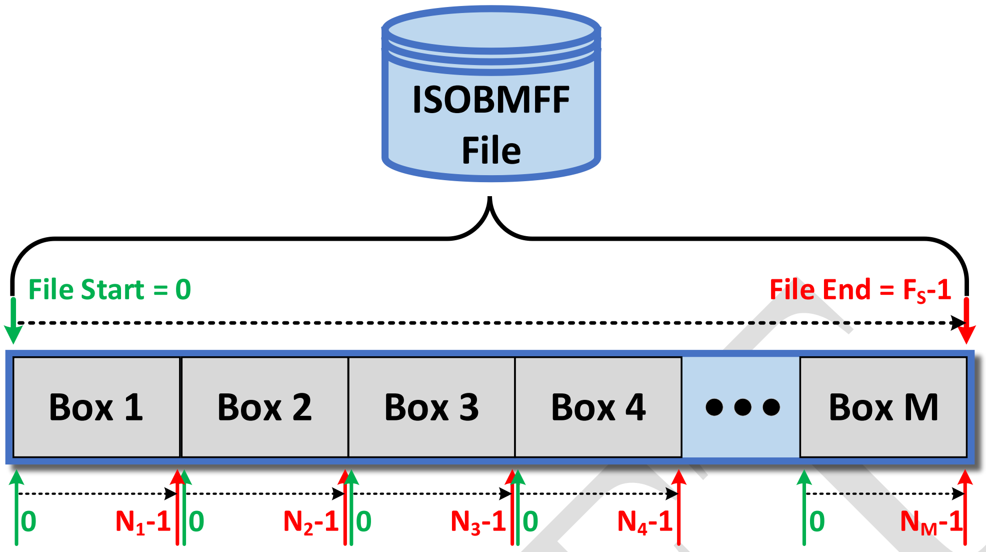


Figure 5: ISOBMFF File Structure

ISOBMFF-Core defines a base set of box types for building a file for an intended application. Beyond this base box type set, the standard allows users to create user-defined boxes if needed.

5.2.2 Box Overview

Each box has two identifiers: a class name and a box type code. The class name is a string of compound words in upper title case (e.g., TitleCase). The box type code is a fixed-length code uniquely identifying the box. Within the documentation, the use of the class name and box type code is synonymous and interchangeable. Within an ISOBMFF-Core file, only the box type code is present.

All boxes have a box header and a payload.

5.2.2.1 Box Header

The box header contains the box size and the box type code. The size specifies the entire size of the box including the box header and payload. The box type code specifies the box's purpose and payload contents, per the box's documentation. Figure 6 illustrates the header and payload structure of a box. The first byte of the header is on the left at zero (0) and extends to the header length $H_i - 1$. The payload immediately follows the header filling out the complete box size of $N_i - 1$.

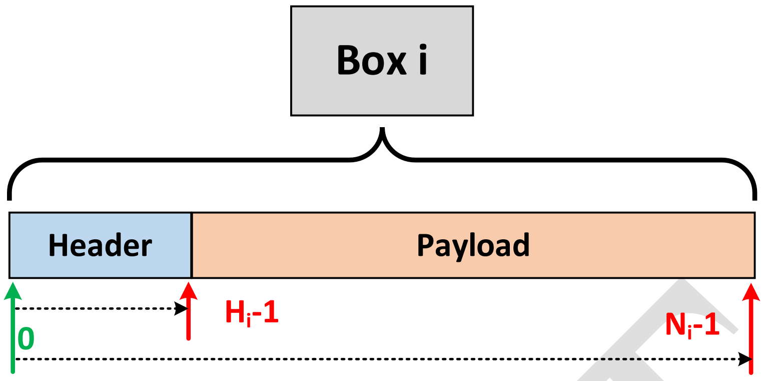


Figure 6: Box Header and Payload

The header contains two pieces of information, the box size and box type code. To provide extensibility, ISOBMFF-Core supports three methods for specifying the size (i.e., number of bytes) of a box (compact, extended, and end-of-box) and two methods for the box type code (compact and extended).

The compact size represents the total number of bytes in a box, where its size is specified using a single uint32 value. The extended size represents the total number of bytes in a box, where its size is specified using a large size uint64 value, along with additional signaling in the header. The end-of-box size signals the payload extends to the end of the file.

The box type code is either a compact 4-byte value or an extended 16-byte value. It is common ISOBMFF-Core practice for each value (i.e., each byte) in the compact 4-byte code to be in the range from 0x20 to 0x7E so each byte represents a utf8 (ISO/IEC 10646:2020) character. The ISOBMFF-Core standard refers to the character-based 4-byte code, as a ‘four-character code’ (4CC). The extended box type is a 16-byte UUID [8] value. When using the extended type code, the compact 4-byte code is set to a 4CC value of “uuid” and an additional 16-byte value appends onto the header.

Given the two different box type codes and three different size techniques, there are six methods ISOBMFF-Core uses to specify a box header:

- Method 1 – (size, box type): the size value contains the number of bytes in the box, and the box type is a 4CC. This is the “compact size” method.
- Method 2 – (size, box type=”uuid”, extended type): the size value contains the number of bytes in the box, the box type is a 4CC equal to “uuid”, and the header includes an extended value (extended_type) of 16 bytes.
- Method 3 – (size=1, box, type, large size): the size value contains the value of one (1), the box type is a 4CC, and the header includes an extended size value (largesize) containing the number of bytes in the box.
- Method 4 – (size=1, box type=”uuid”, large size, extended type): the size value contains the value of one (1), the box type is a 4CC equal to “uuid”, the header includes an extended size value (largesize) containing the number of bytes in the box, and the header includes an extended type value (extended_type) of 16 bytes.

- Method 5 – (size=0, box type): the size value contains the value of zero (0) to indicate the box's payload extends to the end of the file, and the box type is a 4CC. Encoders only use method 5 boxes as the last box in a file.
- Method 6 – (size=0, box type=”uuid”, extended type): the size value contains the value of zero (0) to indicate the box's payload extends to the end of the file, the box type is a 4CC equal to “uuid”, and the header includes an extended type value (extended_type). Encoders only use method 5 boxes as the last box in a file.

Table 1 shows the six methods for defining the size and type of a box. The first column is the method number and its nomenclature; the second column is the size value, which has the name ‘size’ in the ISOBMFF-Core documentation; the third column is the extended size, which has the name ‘largesize’ in the ISOBMFF-Core documentation; the fourth column is the type, which has the name ‘type’ in the ISOBMFF-Core documentation; the last column is the extended type, which has the name “extended_type” in the ISOBMFF-Core documentation. A “N/A” in a table cell means the value is not applicable for the method.

Table 1: Header Methods

Method	‘size’ uint32	‘largesize’ uint64	‘type’	‘extended_type’
1 – compact size	bytes in box	N/A	4CC	N/A
2 – compact size	bytes in box	N/A	“uuid”	16-byte UUID
3 – extended size	1	bytes in box	4CC	N/A
4 – extended size	1	bytes in box	“uuid”	16-byte UUID
5 – payload EOF	0	N/A	4CC	N/A
6 – payload EOF	0	N/A	“uuid”	16-byte UUID

Figure 4 illustrates the byte-wise layout of the six different methods for specifying the header.

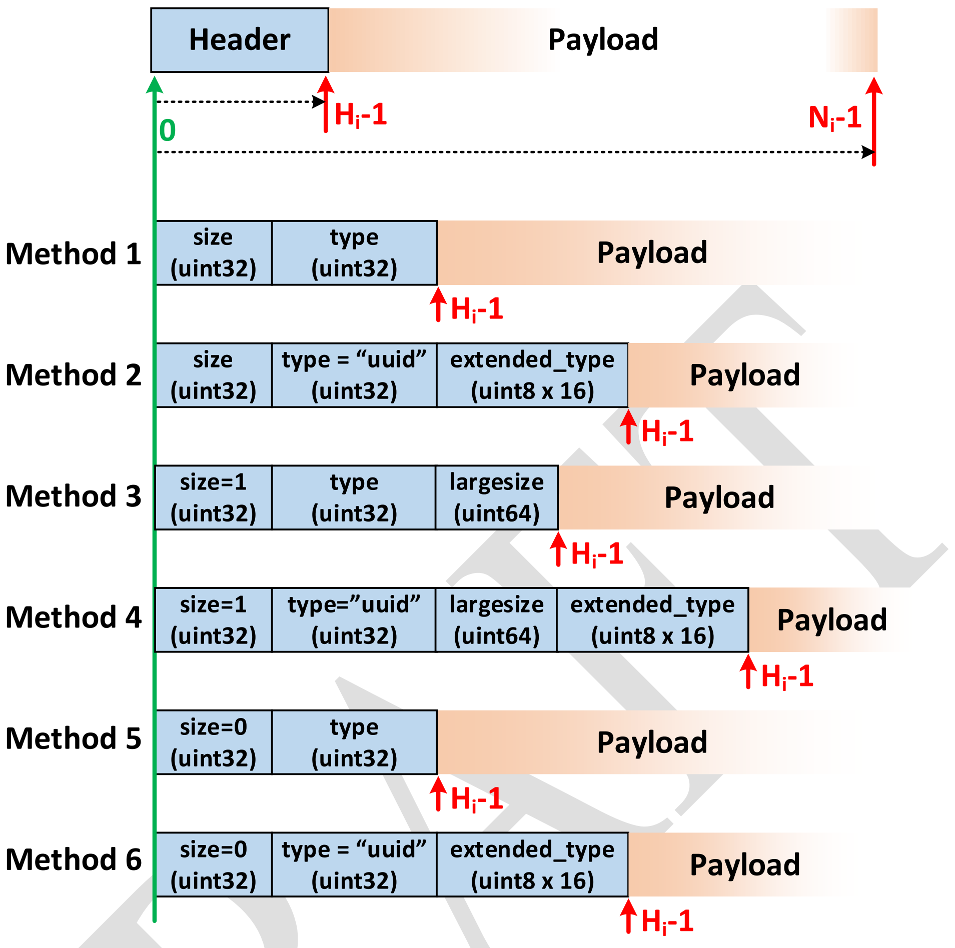


Figure 7: Box Header configurations

5.2.2.2 Box Payload

The box payload is a list of class elements. The documentation for the box uses SDL to define the order and structure of the elements, along with the elements data type, conditional use (if any), and purpose. The class elements may be single values, lists of values, or other boxes. Appendix A provides a summary of SDL, which includes a list of all the data types.

5.2.3 Box Varieties and Names

The ISOBMFF-Core defines additional box varieties and provides names for boxes providing a certain function or capability.

5.2.3.1 FullBox

To support expansion and extensibility beyond the basic "Box," a "FullBox" adds versioning parameters and signaling flags. The versioning and flags support the implementation of new features while maintaining backwards compatibility. The FullBox adds the versioning and flag support after the standard box header.

Figure 8 illustrates the addition of the versioning and flag support for a FullBox. Starting at offset 0, the Basic Header item is one of the six basic box header methods from Section 5.2.2.1. The FullBox header adds a uint8 version number and 24 bits of flag values. Following the FullBox header the box payload begins.

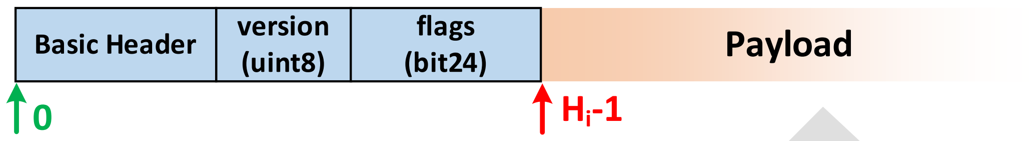


Figure 8: FullBox header

5.2.3.2 Container Box

A container box is any box which includes one or more boxes. Container boxes develop structured hierarchies of boxes. A container box is equivalent to the Super Box terminology found in the JPEG2000 standards.

5.2.3.3 Media Data Box

A media data box holds the content for a presentation.

5.2.4 Box SDL

ISOBMFF-Core describes all boxes with Syntactic Description Language (SDL), which defines the box data structures with a c++ like pseudo code. SDL not only defines the elements and their types, but also includes the order of the elements within the box. As an example, the following SDL from the ISOBMFF document illustrates how the SDL describes box contents. This SDL succinctly describes the same high-level box structures from Section 5.2.2.1 and Section 5.2.3.1.

SDL 1 lists the code for the BoxHeader class, which shows BoxHeader class parameters in orange, the class elements in **green**, and conditional statements in **red**. The SDL element order matches the order of the elements in Figure 7. The two conditionals, “if (size==1)” and “if (boxtype==’uuid’)” determine which Figure 7 method to use.

SDL 1: Box Header

```
aligned(8) class BoxHeader (
        unsigned int(32) boxtype, optional unsigned int(8)[16] extended_type) {

        unsigned int(32) size;
        unsigned int(32) type = boxtype;
        if (size==1) {
                unsigned int(64) largesize;
        } else if (size==0) {
                // box extends to end of file
        }
        if (boxtype=='uuid') {
                unsigned int(8)[16] usertype = extended_type;
        }
}
```

As Figure 7 illustrates all header methods require the size and type elements. The SDL describes both the size and type as 32-bit unsigned integers (uint). With Method 1 the size and type values are set to the number of bytes in a box, and the four-character code (4CC) respectively. With Methods 3 and 4, the size value is one (1) indicating the extended size, and the BoxHeader includes the largesize value, which is an unsigned 64-bit integer. With Methods 5 and 6, the size value is equal to zero (0), indicating the box length extends to the end of the file. With Method 2, 4, and 6, the boxtype value is set to ‘uuid’, and the BoxHeader will contain the extended_type value.

SDL 2 lists the code for the Box class. This class has only one element, the BoxHeader class from SDL 1. The elements of the BoxHeader embed in the Box class and become the first elements of the box.

SDL 2: Box

```
aligned(8) class Box (
       unsigned int(32) boxtype, optional unsigned int(8)[16] extended_type) {

       BoxHeader(boxtype, extended_type);
       // the remaining bytes are the BoxPayload
}
```

SDL 3 lists the code for the FullBoxHeader class, which includes two additional header elements, a version number and set of flags.

SDL 3: FullBoxHeader

```
aligned(8) class FullBoxHeader(unsigned int(8) v, bit(24) f) {
       unsigned int(8) version = v;
       bit(24) flags = f;
}
```

SDL 4 lists the code for the FullBox class. The FullBox class extends the Box class, so now all elements within the Box class are included first in the FullBox. Following the Box class elements, the FullBoxHeader includes its elements.

SDL 4: FullBox

```
aligned(8) class FullBox(
       unsigned int(32) boxtype, unsigned int(8) v, bit(24) f, optional unsigned
       int(8)[16] extended_type)
       extends Box(boxtype, extended_type) {

       FullBoxHeader(v, f);
       // the remaining bytes are the FullBoxPayload
}
```

5.2.5 Box Example

This section provides a simple example to show how ISOBMFF-Core defines a box, then how the box becomes a file of just one box. The example box includes a name, phone number, age, and height. SDL 5 shows the Test1 class extends the Box class and provides the boxtype value as ‘tst1’; this box does not use the extended box type. The Test1 class includes the name and phone elements as utf8strings, the age as an unsigned 8-bit (one-byte) value, and the height as a 32-bit (4-byte) floating point value.

SDL 5: Simple Box Example

```
aligned(8) class Test1() extends Box('tst1') {
        utf8string name;
        utf8string phone;
        unsigned int(8) age;
        float(32) height;
}
```

This SDL defines the specification of the class, but an encoder creates an instance of this box with values for each element in the box. For this example, the instance will use the values: name = "John Doe", phone = "867-5309", age = 37, and height = 1.7 (meters).

For this example, the encoding process starts with the box elements. Table 2 lists the hex encodings of each element in the box. ISOBMFF-Core requires the string encodings include a null value (0x00) to indicate the end of the string.

Table 2: Test 1 Box Encoding Example

Element Name	Value	Hex Encoding	Comment
name	"John Doe"	`4A6F 686E 2044 6F65 00`	Includes a null value (0x00) to terminate the string
phone	"867-5309"	`3836 372D 3533 3039 00`	Includes a null value (0x00) to terminate the string
age	37	`0000 0025`	Unsigned integer in 4 bytes
height	1.7	`3FD9 999A`	IEEE 754 32-bit Floating point value

Combining all the payload element encodings creates the box payload of 26 bytes. With this size known, the encoder can create and prepend the box header. The box is a compact size box (i.e., Method 1) so the boxcode is four bytes and the size is four bytes; therefore, the box size value is 26+4+4 = 34 bytes, or hex value **`0x0000 0022`**. The boxcode value is 'tst1' which becomes hex value **`0x7473 7431`**. Combining the header and payload results in a box with the following binary values in hex:

```
0000 0022 7473 7431 4A6F 686E 2044 6F65 0038 3637 2D35 3330 3900 0000 0025
3FD9 999A
```

5.2.6 Payload Constructs

As Section 5.2.2 describes, a Box contains a header and payload. The payload contains zero or more contiguous elements, where the elements can be different types per the SDL description of each box. Figure 9 illustrates a Box with its Header, Payload, and a view of the Payload as a series of contiguous Elements (Element$_1$ ... Element$_N$). The elements are not necessarily the same size (i.e., number of bytes/bits) and the size of the payload is P_i. The addressing of the payload elements starts at zero on the left to $P_i - 1$ on the right.

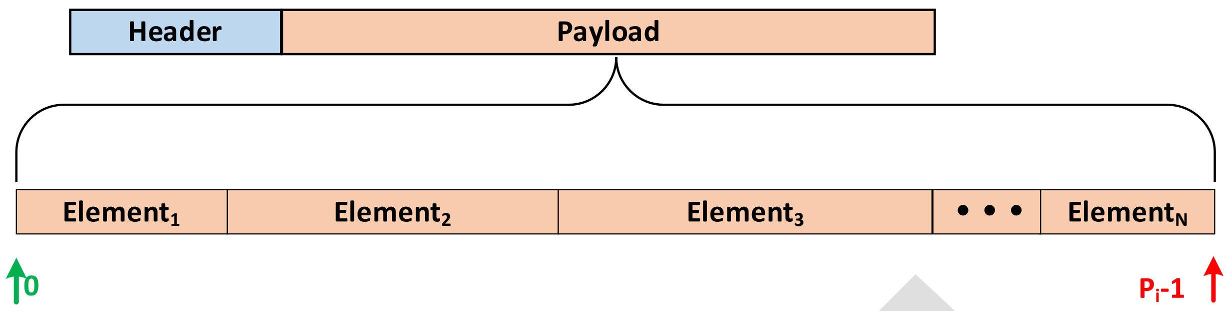


Figure 9: Payload Elements

5.2.6.1 Basic Payload Contents

Elements have different data types and structures: primitive values (e.g., integer, unsigned integer, floating point, bits), primitive lists (e.g., constructed by arrays or for-loops), primitive tuple lists, boxes, and box lists (e.g., constructed by arrays or for-loops).

ISOBMFF-Core defines the boxes using SDL constructs. To aid in the understanding of the boxes and their interactions, this document uses graphical representations of boxes with their data types and structures.

Figure 10 shows an example box graphic with the top stating the box name and 4CC code in bolded underlined text. The remaining part of the graphic is a vertical representation of the (horizontal) payload elements in Figure 9 preserving the order and addressing of the elements. With the Figure 10 graphic every element appears to have the same physical space (i.e., byte size); however, each element has their unique size per Figure 9 (i.e., the length of the elements is obscured). The SDL for the box describes the order and conditionality of the elements within the box. The box's name and elements within the box, along with the SDL, enable a reader to determine the correct syntax and semantics of each element.

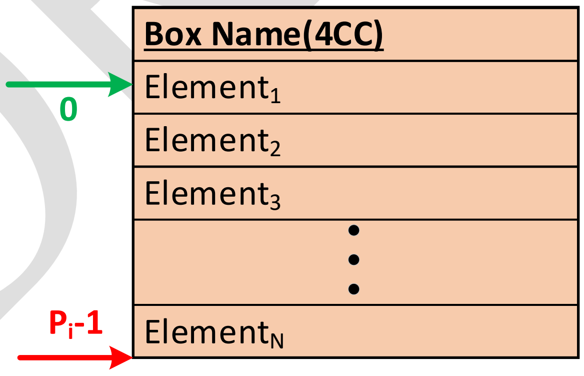


Figure 10: Graphical representation of a box

In this document the different data types and structures have different visual representations within the diagrams showing:

- primitive types without element-line separation,
- primitive tuples in parenthesis without element-line separation,

- primitive lists with bullets prefixing the list items and without element-line separation,
- primitive tuple lists with bullets in front of each parenthesized tuple, and without element-line separation,
- boxes shifted right with upper and lower element lines, and
- box lists shifted right plus curly brace on right with upper and lower element lines.

Figure 11 illustrates examples of diagrams with primitive elements. Figure 11 (1) shows the primitive elements in the payload as SDL would list them. Figure 11 (2) shows a primitive element, a tuple, followed by more primitive elements. A tuple is shorthand for small groups of related items, e.g., position and size of a data chunk. Tuples are not in the SDL; however, they are in this document only to aid visualization and provide logical connectivity. Figure 11 (3) shows a primitive element (Element$_1$) prior to a list of elements; in many cases the element before a list is the list count. Figure 11 (4) shows a primitive element (Element$_1$) before a list of tuples; after the tuple list there is another primitive element (Element$_N$). In this example, each tuple contains three primitive tuple values, (T_1, T_2, T_3).

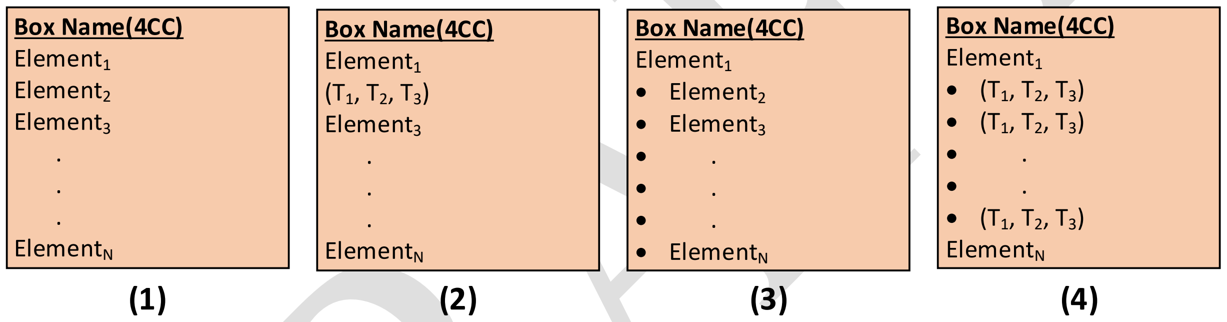


Figure 11: Diagram Examples with Primitive Elements

Figure 12 illustrates diagram examples showing boxes contained within boxes. Figure 12 (1) contains a primitive element (Element$_1$), box A (indented), box B (indented), and primitive element (Element$_3$). A standard (e.g., ISOBMFF-Core) defines SDL to state which boxes to use and dictates the order of the elements within the box. In the diagram, the area next to the indented boxes is only for visualization and does not represent any additional data, framing, or space in the parent box. The two boxes are of different types (type A and B), and each has its own set and type of elements, which can include primitives, lists or other boxes. Figure 12 (2) shows a primitive element (Element$_1$), an embedded box list (as indicated by the bracket to the left), and primitive element (Element$_3$). With this example the box list contains the same list-item type, i.e., Box Name A; however, depending on the list definition in the SDL the list-items may have different types.

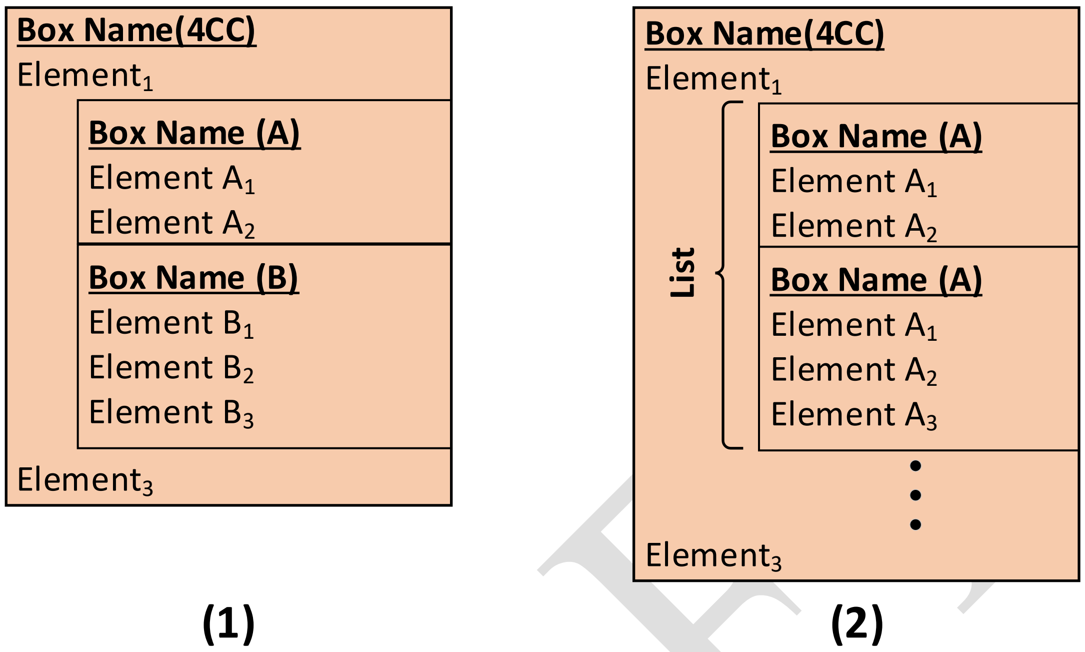


Figure 12: Diagram Examples with Box Elements

SDL allows for constructing many different types of box constructs, which this document will represent as the combination of the different diagram types in Figure 11 and Figure 12.

5.2.6.2 Box and Data Relationships

The box-within-box structure enables the construction of data hierarchies. This allows file producers to directly relate data constructs together via data locality (i.e., parent/child relationships); however, hierarchies are not enough to link information together. The ability to reference other children or parents from any other child or parent is necessary to enable simple and complex configurations of data (e.g., media).

There are two methods for defining relationships beyond the data hierarchy, logical and physical referencing. Some boxes use logical referencing by defining an identifier (e.g., item_id) which other boxes can refer to forming a relationship between the identified box (destination) and reference-er box (source). Some boxes use physical references, which are byte offsets from an origin point. The origin point can be the beginning of the ISOBMFF-Core file, a box’s payload, or a specific location within a box. Additionally, the origin point can be the start of another file, e.g., a different ISOBMFF file or alternate file type.

Figure 13 illustrates two logical relations as dotted line with arrows. The first relationship is between boxes A and B, where A references B. Box B is the destination by defining a box_id with the value 7; box A is the source with a primitive element logical_ref1 with the value 7. The boxes’ SDL define what the element names are (i.e., box_id and logical_ref1 are not actual names in ISOBMFF-Core, these are just examples) and the box documentation describes the meaning of the relationship. Each box relationship source element has a specific purpose, so the line initiates at the source box element.

The second relationship is between box A and box E, where A references E. This example illustrates the destination of the reference is a box embedded in a list.

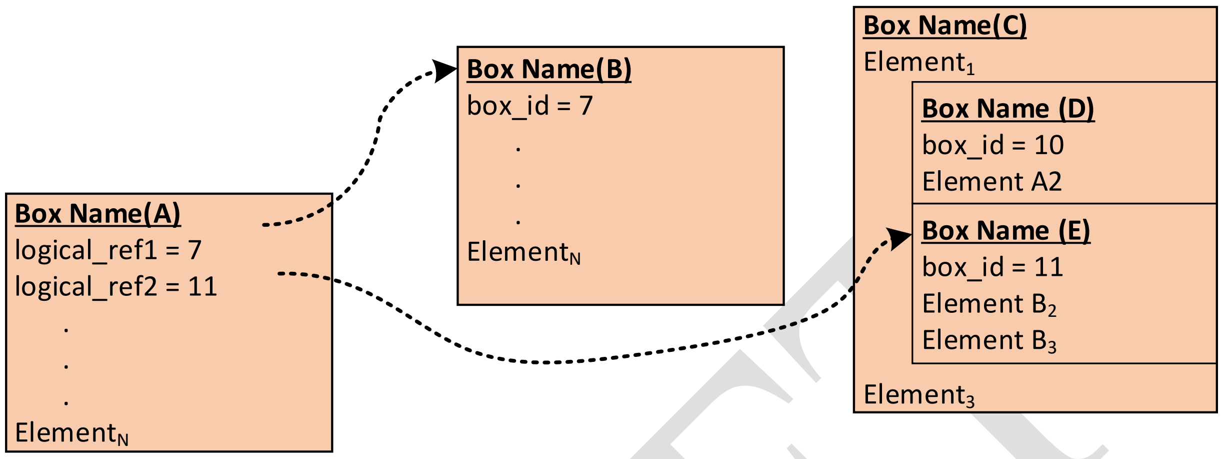


Figure 13: Logical Relationships

Figure 14 illustrates physical relationships between box A and portions of box C, where A references a subset of C. This figure shows box A containing two physical references, with each a two-tuple containing a starting offset (S_i) and length (L_i). In this illustration, Box C contains one element of data for the whole box. In this example the origin of the reference is the start of box C. The phy_ref1 in box A defines a starting offset, which is the number of bytes from the origin of Box C to the desired data block; the length is the number of bytes in the data block. The figure illustrates the data block as a shaded region in Box C. The SDL dictates how to determine the origin type, such as a start of box payload or start of file.

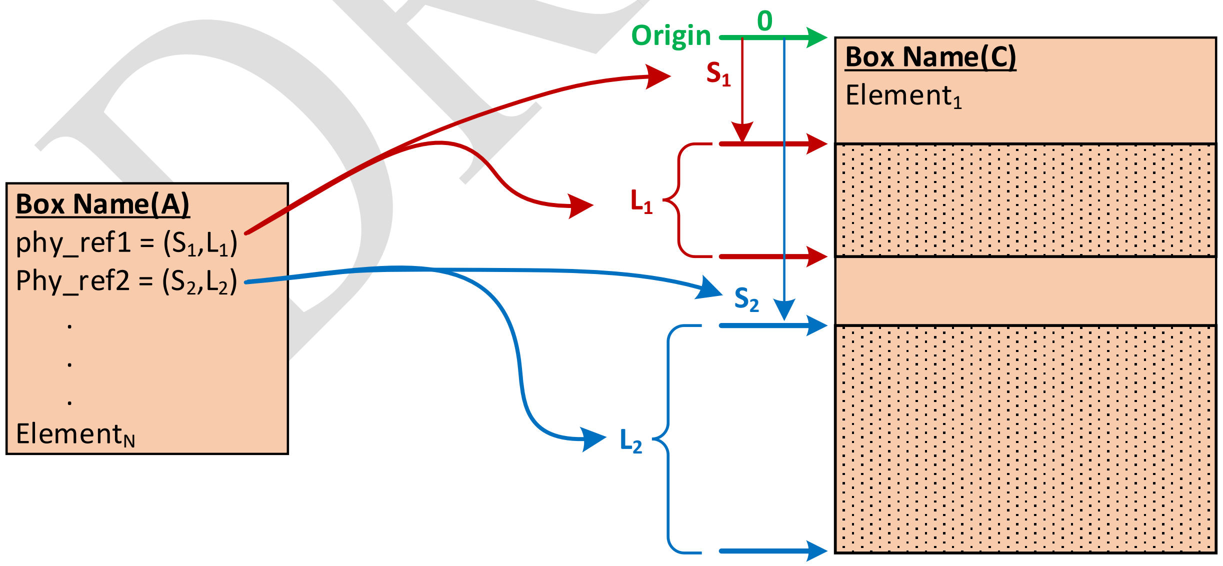


Figure 14: Physical Relationships

In this document the details of where a box starts and its length are not important to show, so the figures will only show a solid black reference line from the source's physical reference element to some part of the destination box. Figure 15 provides an example of how this document will illustrate the physical references.

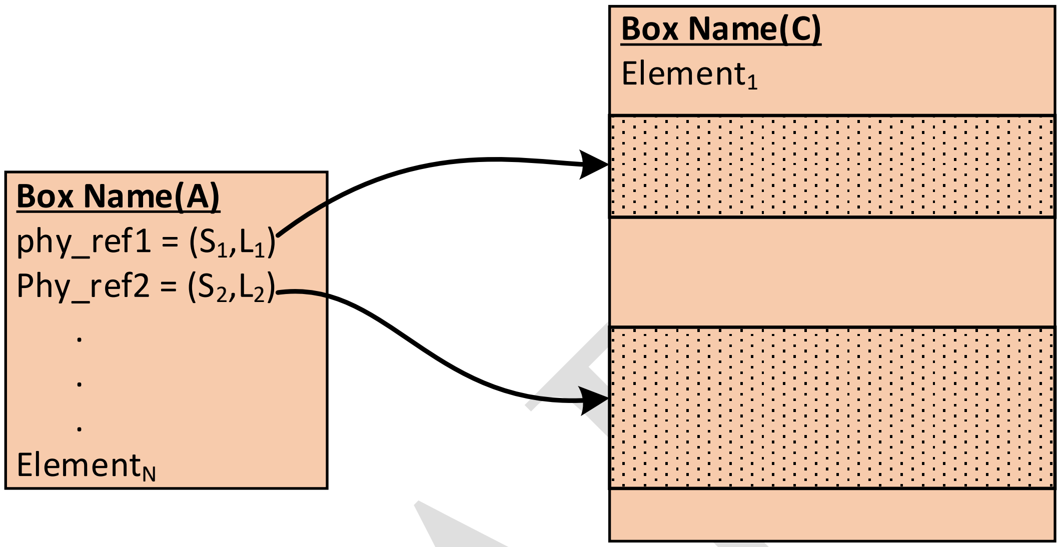


Figure 15: Physical references

When a box diagram includes the outermost level of the file – the file rectangle visualization includes a blue border and blue background. Figure 16 illustrates a file box with the blue border.

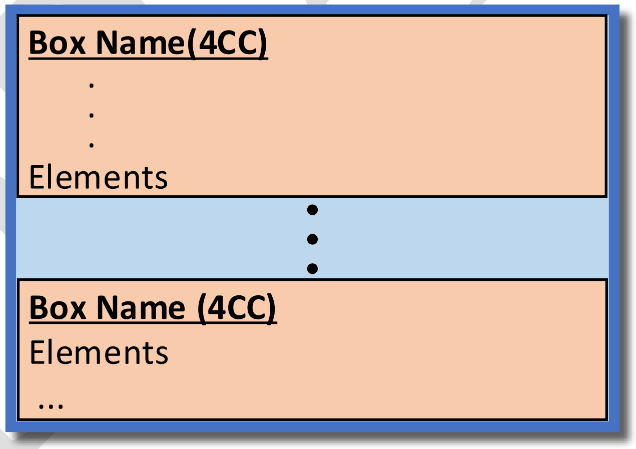


Figure 16: Box list showing outer file

5.2.7 Box Details

This document uses a common table format to provide reference information to ISOBMFF boxes. As Table 3 illustrates, the Box Reference Table has three columns: Name, Purpose, and Reference. The Name column provides both the class name and the box's 4CC (in parenthesis). The Purpose column provides a short description and whether the box is mandatory; optional boxes have no comment on their optionality. The Reference column states where to go to get more information, ISOBMFF-Core means, refer to the ISOBMFF-Core standard, using the box name in the first column.

The Table 3 example lists one row or box. The name column contains the box name as “ClassName” and its 4CC is “clnm”; the purpose column contains a brief description along with the text “(Mandatory)” if the box is required; the reference column states where to go for more information, either a section number within this document or the ISOBMFF-Core document.

Table 3: Example Box References

Name	Purpose	Reference
ClassName (clnm)	Short Description or comment (Mandatory) or non if optional	Section Number or ISOBMFF-Core

5.3 Base Box Structure in ISOBMFF-Core

There is one type of application the ISOBMFF-Core standard supports, the storing of timed media, e.g., video, audio. The ISOBMFF-Core allows for different codecs and thus different storage and usage patterns.

Regardless of the storage pattern, all current implementations of ISOBMFF-Core have a base box structure and requirements. Specific codecs extend the base box structure by adding additional boxes as necessary to meet the codec’s requirements. Early versions of ISOBMFF-Core did not have the base requirements, so the base box structure may not apply to legacy files; the legacy differences are beyond the scope of this document. This document defines the base box structure for typical ISOBMFF-Core applications which have one or more timed media within a file.

The base box structure contains the boxes: FileTypeBox, MovieBox, and one or more MediaDataBoxes. An ISOBMFF-Core file is a list of boxes of varying type, but for all applications there is only one required box, the FileTypeBox. The FileTypeBox identifies the file’s media types, allowing reader software to properly allocate the right resources to read, interpret, and process the file contents. ISOBMFF-Core files require the FileTypeBox to be at the beginning of the file, possibly the first box of the file. There is not a specific order of the other boxes in the file.

Timed media has two parts: the information about the media and the media data itself. Each ISOBMFF-Core file uses a single MovieBox to provide information about the timed media. Applications embedding timed media within an ISOBMFF-Core file require at least one MediaDataBox. The ISOBMFF standard does allow hosting media in alternate files.

Applications store all the various media types (e.g., video, audio) in one or more MediaDataBoxes. Other boxes (e.g., MovieBox) use physical relationships to reference the media in the MediaDataBox.

Figure 17 illustrates an ISOBMFF-Core file with the FileTypeBox defining the brands as the first box in the file, followed by the MovieBox which provides media information, and ending with one MediaDataBox which provides the actual media-encoded data. The MovieBox contains elements (i.e., child boxes) that eventually reference data in the MediaDataBox using physical references.

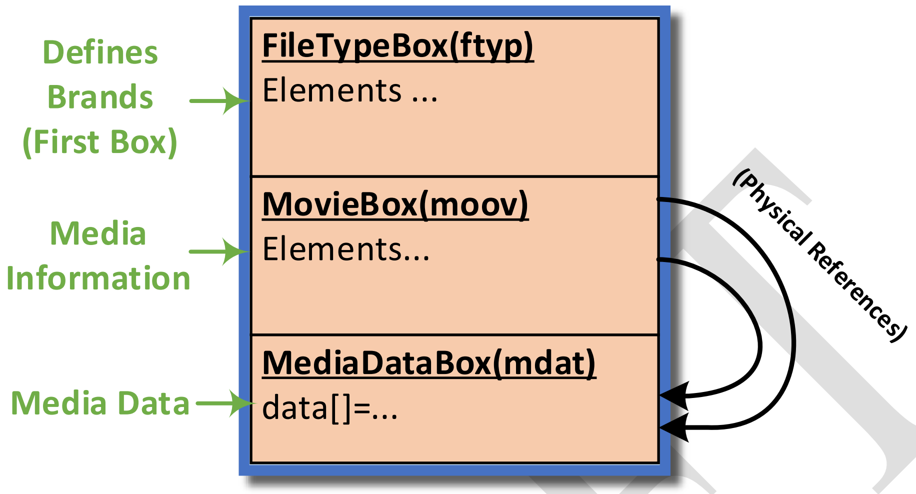


Figure 17: File with FileTypeBox, MovieBox, and MediaDataBox

5.4 File Information Details

File information includes the definition of brands and progressive downloading information.

The base box structure describes the requirement for including brand information in every file. There are two methods for including brand information using the FileTypeBox and OriginalFileTypeBox. The FileTypeBox defines the brands for the data in the file (see Section 5.4.1). The OriginalFileTypeBox defines brands for the file where file compression or encryption have disturbed the structure of the data, such that, putting the brands into the FileTypeBox will not represent the data anymore. The OriginalFileTypeBox enables readers to understand what data is within the compressed or encrypted file without needing to decompress or decrypt the data.

The MetaBox provides a broad range of metadata.

Table 4 provides all the File Information Box References.

Table 4: File Information Box References

Name	Purpose	Reference
FileTypeBox (ftyp)	The brands the ISOBMFF file adheres to. (Mandatory)	Section 5.4.1
OriginalFileTypeBox (otyp)	The brands the ISOBMFF contents adheres to after transformations (e.g., decompressed)	ISOBMFF-Core
MetaBox (meta)	General File-Level metadata	ISOBMFF-Core

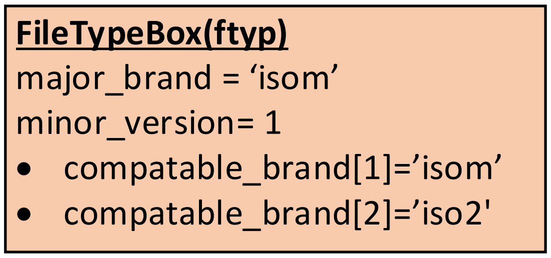
5.4.1 FileTypeBox

The FileTypeBox contains three elements: major brand, minor version, and list of compatible brands. Brands identify the specifications in use within the file; they indicate requirements for producers when generating files and for readers when decoding, interpreting, and presenting content. Each brand invokes rules for which ISOBMFF-Core boxes applications need to use. The brand value is a pre-registered 4CC. Annex E of ISO/IEC 14496-12 lists one set of brands; other ISO documents provide other brand lists.

The major brand indicates the primary requirements to a reader application. The major brand indicates which specification represents the 'best use' of a file. The minor version for the major brand is informative only and may aide in debugging and file inspection. The version does not affect conformance to the major brand. Reader applications may ignore the minor brand.

The compatible brands list provides a set of specifications to which a file is conformant. Typically, the compatible brands list includes the major brand. To enable all features and content within a file, a reader application must implement all features for the specific brands in the compatible brands list – an ISOBMFF-Core requirement.

Figure 18 illustrates a FileTypeBox with the major brand of 'isom' (from AnnexE). The isom brand indicates the file may include up to 47 different box types to describe the media. Applications can ignore the minor version in this example. The compatible_brand list re-lists the 'isom' brand and includes the 'iso2' brand. The 'iso2' brand indicates the file may include an additional 13 different boxes beyond what the 'isom' brand indicates. In addition to the box lists for the brand, the Annex lists other brand requirements.



FileTypeBox(ftyp)

major_brand = 'isom'

minor_version= 1

- compatable_brand[1]='isom'
- compatable_brand[2]='iso2'

Figure 18: FileTypeBox Example

5.5 Movie Structure Details

The movie structure is the hierarchal administrative information necessary to play the presentation. ISOBMFF-Core provides both fragmented and non-fragmented movie structures. This document does not address the fragmented structure. The root box of the non-fragmented hierarchy is the MovieBox.

5.5.1 MovieBox

The MovieBox is a container box which provides Movie Information and a list of Tracks.

The Movie Information includes the MovieHeaderBox and MetaBox. The MovieHeaderBox provides the movie's creation time, modification time, timescale, duration, playback rate, volume, transformation and more. The MetaBox contains information about the movie, but unrelated to the timed media.

The list of tracks is a series of TrackBoxes containing information about each track in the movie. Each track contains its own set of boxes. Figure 19 shows the Movie Information in the overall hierarchy of the MoveBox.

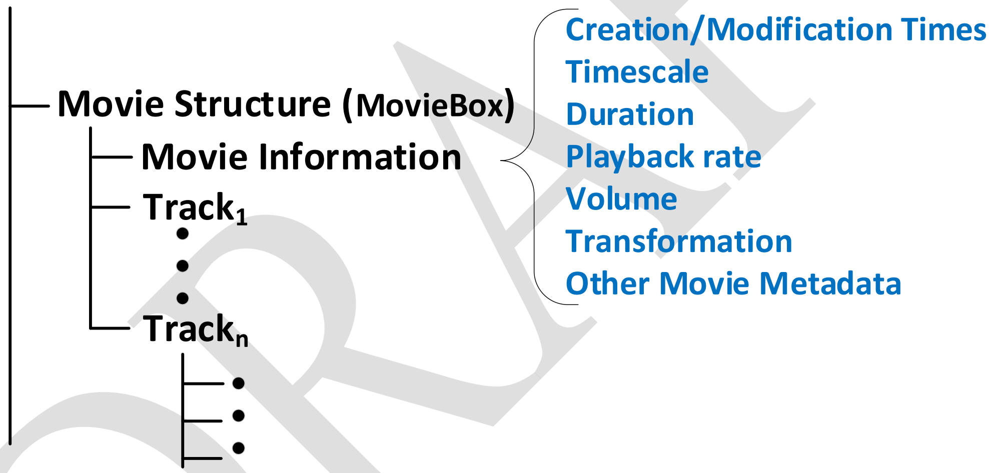


Figure 19: Movie Information

Table 5 lists the boxes the ISOBMFF-Core standard defines for the MovieBox.

Table 5: MovieBox Children Box References

Name	Purpose	Reference
MovieHeaderBox (mvhd)	Defines overall information about the media. This box is mandatory.	ISOBMFF-Core
MetaBox (meta)	Container box encapsulating metadata about the media.	ISOBMFF-Core
TrackBox (trak)	Container box for one or more track descriptions. Mandatory to have at least one box.	Section 5.5.1.1

A movie, and each track within a movie, has a **timescale** defined as “a number of ticks per second”, as well as the **duration** of the movie, indicated as “a number of ticks.” Timescales in tracks, in general, are different based on the sampling rate of the media carried. The following example is a video/audio clip with the timescales and durations of the movie and the media:

Given: Movie timescale = 1000 ticks per second, and Movie duration = 3721 ticks
∴ The length of the movie = 3721 ticks / 1000 ticks-per-second = 3.721 seconds.

Given: Video track timescale = 90000 ticks per second, and track duration = 334882 ticks
∴ The length of the video track = 334882 / 90000 = 3.721 seconds
For a count of frames = 111, the frame rate = 111 samples / 3.721 sec = 29.83 FPS

Given: Audio track timescale = 48000 ticks per second, and track duration = 177152 ticks
∴ The length of the audio track = 177152 / 48000 = 3.69 seconds

The timescales within a file are relative and define the relationships amongst the different media contributions. There is no inherent synchronization to an absolute time source, such as wall clock. Timescales for tracks are typically different based on the sampling rate of the media carried. The duration of a movie is usually the duration of the longest track in the file.

5.5.1.1 Track

A Track is a collection of samples representing a time-period within the movie. The samples are all the same type within the track, i.e., all video or all audio, etc. Tracks may have associations with other tracks, such as an audio track associated with a video track. Each track represents a complete set of samples, but for playout, a track edit list may instruct the player to trim, repeat, or shuffle samples.

Each track contains Track Information, Track Associations, an Editing List, and Media/Sample Details. The Track Information includes track type, identifier, creation/modification times, duration, dimensions for imagery or text, audio volume, transformation matrix and other metadata. Track Associations define relationships between tracks either using Track Referencing or Track Grouping. The Editing List is a series of manipulations to the media to produce the desired presentation. Figure 20 shows the Track Information, Track Associations, and Edit List in the hierarchy.

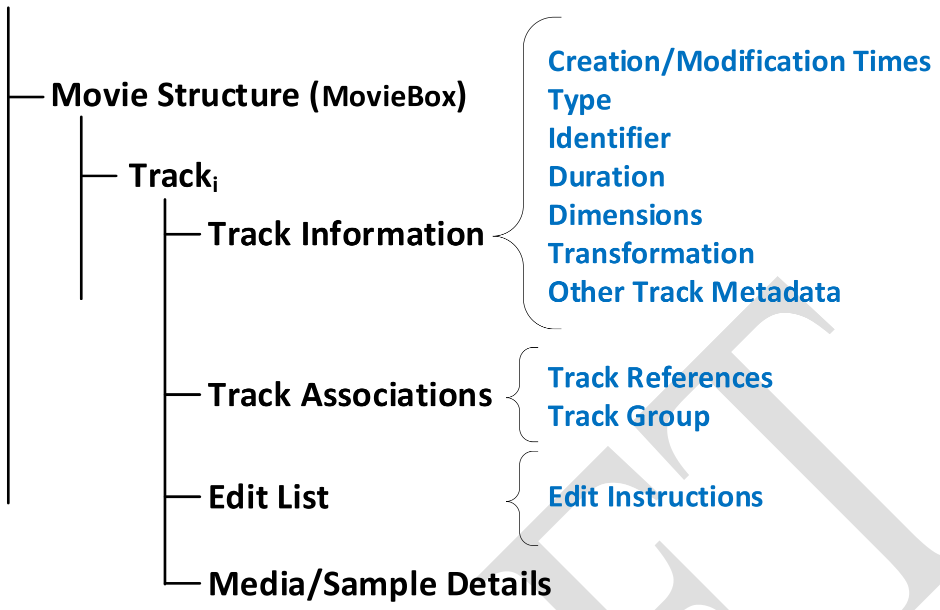


Figure 20: Track Information and Track Association

Table 6 lists the Track Information boxes and references to more detail.

Table 6: Track Information Box References

Name	Purpose	Reference
TrackHeaderBox (tkhd)	Specifies wholistic track information including track identification for logical references.	ISOBMFF-Core
MetaBox (meta)	A track's untimed metadata	ISOBMFF-Core

Table 7 lists the Track Associations boxes.

Table 7: Track Associations Box References

Name	Purpose	Reference
TrackReferenceBox (tref)	Track Association: Defines dependency relationships between tracks.	ISOBMFF-Core
TrackGroupBox (trgr)	Track Association: Defines a set of tracks sharing a particular characteristic or relationship.	ISOBMFF-Core

Table 8 lists the Track Edit List box.

Table 8: Track Edits Box References

Name	Purpose	Reference
EditBox (edts)	Series of manipulations to the media to produce the desired presentation.	Section 5.5.1.1.1

Table 9 lists the MediaBox box.

Table 9: Media Box Reference

Name	Purpose	Reference
MediaBox (mdia)	Definition of media and sample information.	ISOBMFF-Core

The MediaBox contains boxes which includes all the Media/Samples Details. The Media/Sample Details provide information about the media in the file such as the codec name/type, sample timing, sample data locations, and more. Figure 21 shows the Media/Sample Details information in the track hierarchy.

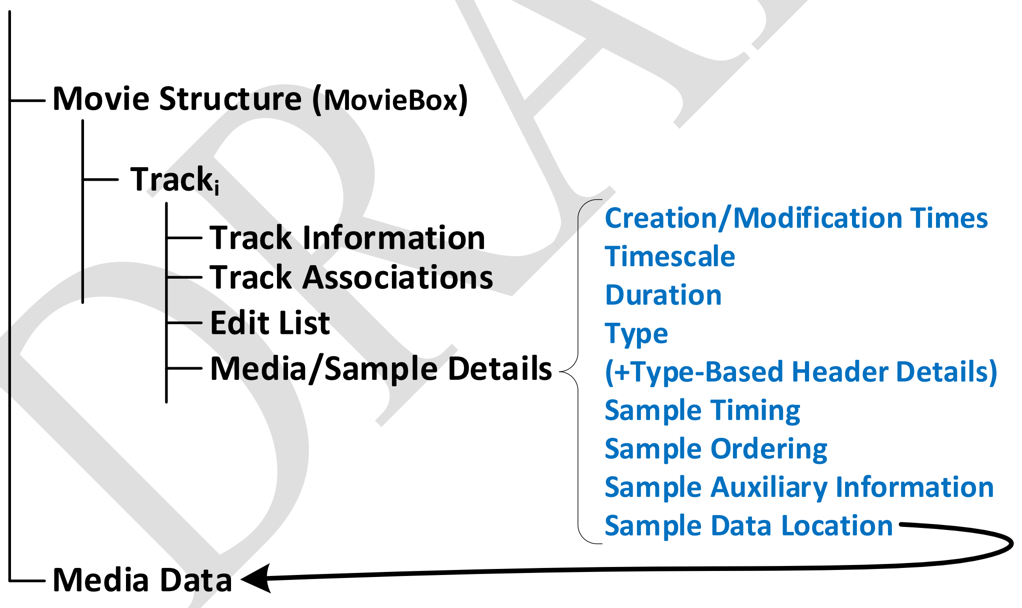


Figure 21: Media/Sample Details

Table 10 lists the boxes of the Media/Samples details. The MediaInformationBox contains the Type-Based Header and sample information.

Table 10: Media/Sample Details Children Box References

Name	Purpose	Reference
MediaHeaderBox (mdhd)	Declares overall information and characteristics of the media in the track.	ISOBMFF-Core
HandlerBox (hdlr)	Declares the type of the media in the track.	ISOBMFF-Core
MediaInformationBox (minf)	Container for describing characteristics about the media.	ISOBMFF-Core

When the encoder defines the type of encoding for the media, the encoder includes an appropriate header for the encoding type. Table 11 lists the header boxes for the various types of media.

Table 11: Media Types Box References

Name	Purpose	Reference
VideoMediaHeaderBox (vmhd)	Media header information if the track is a video track.	ISOBMFF-Core
SoundMediaHeaderBox (smhd)	Media header information if the track is an audio track.	ISOBMFF-Core
HintMediaHeaderBox (hmhd)	Media header information if the track is a hint track.	ISOBMFF-Core
SubtitleMediaHeaderBox (sthd)	Media header information if the track is a subtitle track.	ISOBMFF-Core

The Media/Sample Details is responsible for providing the locations of all the media resources. Locations consist of two parts: the origin and offset, see Section 5.2.6.2. The origin is either a location within the ISOBMFF or a separate file. The offset is a byte offset from the origin. Table 12 lists the DataInformationBox which defines the origin locations.

Table 12: Data Information Box Reference

Name	Purpose	Reference
DataInformationBox (dinf)	Defines the locations of the media data.	ISOBMFF-Core

A track consists of a list of samples, each of which has its own time and other information. Table 13 lists the SampleTableBox which contains all the information about sample timing, sample ordering, sample size, sample location, and sample auxiliary information.

Table 13: Data Information Box Reference

Name	Purpose	Reference
SampleTableBox (stbl)	Table of information about the samples in the media box.	ISOBMFF-Core

Within the track, samples have a decode time, composition time, size, location, and type. Table 14 lists the boxes within the SampleTableBox providing this information. This information provides the basis for the timing model interlinking the decode, composition, and presentation times.

Table 14: Media Information Children Box References

Name	Purpose	Reference
SampleDescriptionBox (stsd)	Provides detailed coding type information and any coding initialization information.	ISOBMFF-Core
TimeToSampleBox (stts)	Provides a mapping from decoding timestamp to sample number.	Section 5.5.1.1.1
CompositionOffsetBox (ctts)	Provides offset time from decode time to the composition time.	Section 5.5.1.1.1
SampleToChunkBox (stsc)	Provides a mapping between samples and data chunks.	Section 5.5.1.1.2
SampleSizeBox (stsz)	Defines the size or sizes of the samples.	ISOBMFF-Core
ChunkOffsetBox (stco)	Lists the starting locations (offsets) of chunks.	Section 5.5.1.1.2
SampleDependencyTypeBox (sdtp)	Lists dependency information for each sample.	ISOBMFF-Core
SampleToGroupBox (sbgp)	Mapping from sample to Group.	ISOBMFF-Core
SampleGroupDescriptionBox (sgpd)	Describes the Sample Group.	ISOBMFF-Core

In addition to the time and space information, the SampleTableBox optionally includes auxiliary information on a per samples basis. Table 15 lists the boxes for defining the type and location of the metadata.

Table 15: Media Information Children Box References

Name	Purpose	Reference
SampleAuxiliaryInformationSizesBox (saiz)	Defines the type and size of the auxiliary information.	ISOBMFF-Core
SampleAuxiliaryInformationOffsetsBox (saio)	Defines the offsets in the file to locate the auxiliary information.	ISOBMFF-Core

5.5.1.1.1 Sample Timing and Ordering – The ISOBMFF-Core Timing Model

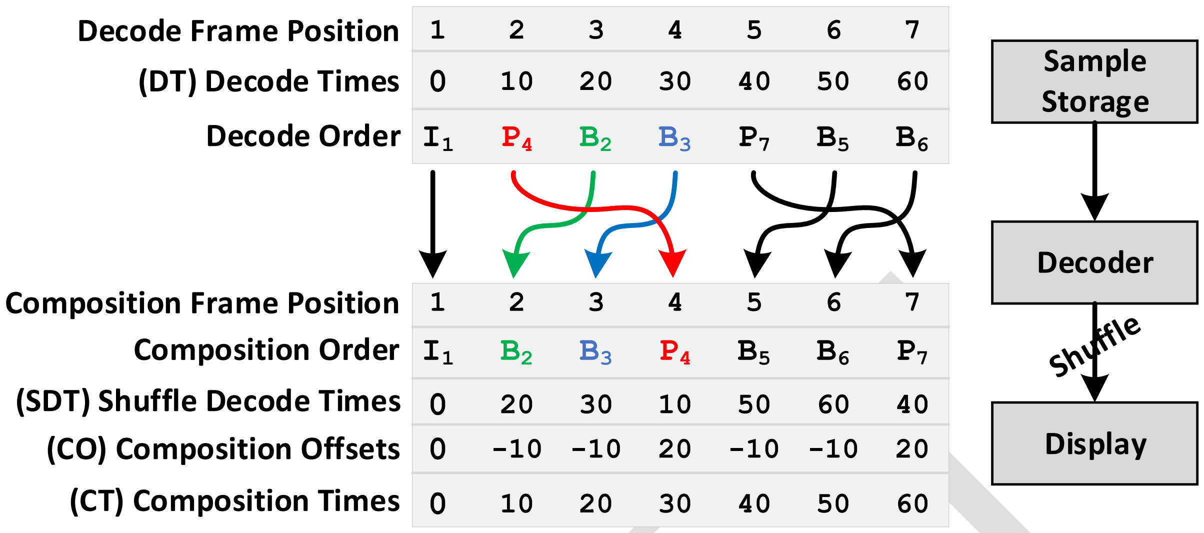
The ISOBMFF-Core timing model uses three timelines, the **decode timeline**, **composition timeline**, and **presentation timeline**.

To make use of predictive and bidirectional coding techniques, many video codecs order the compressed video frames differently than the original uncompressed frames order. Decoders rely on an image frame (I frame) to decode a predictive frame (P frame). Furthermore, the decoder relies on both an I frame and P frame to decode a bidirectional frame (B frame). For example, a decoder may display a decoded MPEG frame sequence as $I_1B_2B_3P_4B_5B_6P_7$, where the number indicates the display order. To decode the B_2 and B_3 frames the decoder needs to decode both the I_1 and P_4 frames first, and to decode P_4, the decoder needs to decode I_1 first. Decoding, B_5, B_6 and P_7 follow similar constraints; thus, the decode order is $I_1P_4B_2B_3P_7B_5B_6$.

The **decoding timeline** assigns a timestamp to every sample in the order the decoding occurs. The decoding timeline in each track is zero-based, so the first sample's decode time is zero (0). The **composition timeline** is the display time for each sample in the track. The composition timeline in each track is zero-based, but the track's first sample may start later than zero. The sample in the track with the smallest composition time is the first sample the decoder will display. The **presentation timeline** uses edit lists to adjust when to display the frame, see Section 5.5.1.1.1.1. When the decoding order matches the composition order, there is no need for specifying the composition order in the track.

Figure 22 illustrates an example of how the decode order, decode timing, composition order, and composition timing works together. The right side of the figure shows the processing flow from sample storage (in an ISOBMFF-Core file) to a decoder, then to display via a "shuffle" operation. ISOBMFF-Core requires the storage of samples in decode time order. The first line lists the Decode Frame Position in the Sample Storage. The next line lists the Decode Times (DT) for each sample (frame).

The Decode Order line shows the IPB order of the frames which is the same as the example above. After decoding, the resulting decoded frames need reordering before display. The arrows show the "shuffling" into the Composition Order, I_1 displays first, B_2 displays second (green), B_3 displays third (blue), P_4 displays fourth (red), etc. Below the composition order, the figure shows the original Shuffle Decode Times (SDT) after shuffling. The Composition Offsets (CO) line is the difference between the Decode Times and Shuffled Decode Times, for the given frame position, i.e., $CO = DT - SDT$. The final line is the Composition Times (CT) which provides both the time and order of the frames display, $CT = SDT + CO$.



Decode Frame Position	1	2	3	4	5	6	7
(DT) Decode Times	0	10	20	30	40	50	60
Decode Order	I_1	P_4	B_2	B_3	P_7	B_5	B_6
Composition Frame Position	1	2	3	4	5	6	7
Composition Order	I_1	B_2	B_3	P_4	B_5	B_6	P_7
(SDT) Shuffle Decode Times	0	20	30	10	50	60	40
(CO) Composition Offsets	0	-10	-10	20	-10	-10	20
(CT) Composition Times	0	10	20	30	40	50	60

Figure 22: Decode and Composition Code Example

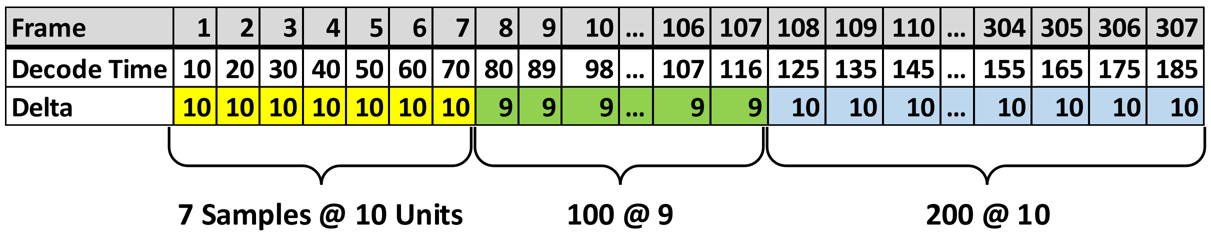
ISOBMFF-Core stores the **Decode Time** (DT) and **Composition Offsets** (CO) within each track.

For **Decode Times**, instead of explicitly storing the decode time for each sample, the TimeToSampleBox stores the decode times as a list of sample time differences. Typically, video decodes and displays frames at a constant rate, so ISOBMFF stores just the common difference, or delta, between each frame. For example, in Figure 22, the difference between each Decode Time (DT) is 10 units; instead of storing seven separate values (i.e., 0, 10, …60), the TimeToSampleBox stores two values, the number of samples (7) with the same delta, and the delta value, which is 10. The delta may change at any given time, so ISOBMFF stores a list of sample counts and delta values for each track. Figure 23 provides an example TimeToSampleBox.

TimeToSampleBox(stts)
entryCount = 3
- (sample_count=7, sample_delta=10)
- (100, 9)
- (200,10)

Figure 23: TimeToSampleBox Example

Figure 24 illustrates a track with 307 samples in three decode time groups. The first group matches the example from Figure 22 with 7 samples having a delta of 10. Following the first group, the track has another 100 samples with the delta of 9, then for another 200 samples the delta is back to 10. The resulting decode time list is three tuples of sample count and sample delta, (count, delta): (7, 10), (100, 9), (200, 10).



Frame	1	2	3	4	5	6	7	8	9	10	…	106	107	108	109	110	…	304	305	306	307
Decode Time	10	20	30	40	50	60	70	80	89	98	…	107	116	125	135	145	…	155	165	175	185
Delta	10	10	10	10	10	10	10	9	9	9	…	9	9	10	10	10	…	10	10	10	10

7 Samples @ 10 Units
100 @ 9
200 @ 10

Figure 24: Decode Time Example

For **Composition Offsets**, instead of explicitly storing the composition offset for each sample the CompositionOffsetBox stores the composition offsets in a compact form. With the list of composition offsets, when one or more successive sample's offsets are identical, the encoder keeps a repetition count for that sample offset. The result is an ordered tuple list of composition sample count and sample offset (count, offset). For example, in Figure 22, the composition offsets are, 0, -10, -10, 20, -10, -10, 20, which becomes tuples (1,0), (2, -10), (1, 20), (2, -10), (1, 20). The first tuple indicates there is one 0 offset; the second tuple indicates there are two -10 offsets, etc.

When the decoding order matches the presentation order all the composition offsets are zero and there is no need to store them in the track data.

5.5.1.1.1.1 Presentation Timeline

The presentation timeline adjusts the track's composition time to determine what the track's sample output should be. The adjustments are in the form of an ordered list of edits. Each edit declares a section of the composition timeline to include in the presentation timeline. A single edit can encompass the whole composition timeline or any portion of the timeline down to a single sample.

Each edit record is a three-tuple with edit duration, media time, and media rate: tuple = (edit_duration, media_time, media_rate). The edit duration is the size of the edit in number of time units of the composition timeline. The media time is the starting point of the edit on the composition timeline. The media rate is the playback speed of the edit on the presentation timeline.

Figure 25 illustrates the process of an edit from the composition timeline to the presentation timeline. The top line of the figure shows the composition timeline in blue, with its starting and ending points. The figure illustrates the starting value of the composition timeline may be greater than zero. The green line represents the presentation timeline with its starting and ending points. The length of the presentation timeline comes from the aggregation of the four edits of the composition timeline. The first edit starts at time unit t_0 for a duration of d_0 time units (in red) with a rate of 1.0 or in tuple form, $(t_0, d_0, 1.0)$. The first edit becomes the first section of the presentation time. The second edit is $(t_2, d_2, 1.0)$ which becomes the second section of the presentation time; this edit shows the composition timeline source may come from any section of the composition timeline. The third edit is $(t_1, d_1, 0.666)$ which becomes the third section of the presentation time; this edit shows a rate of 0.666, which increases the duration of the presentation timeline section by 50%, for that section. The fourth edit is $(t_3, d_3, 1.0)$, which becomes the final section of the presentation timeline.

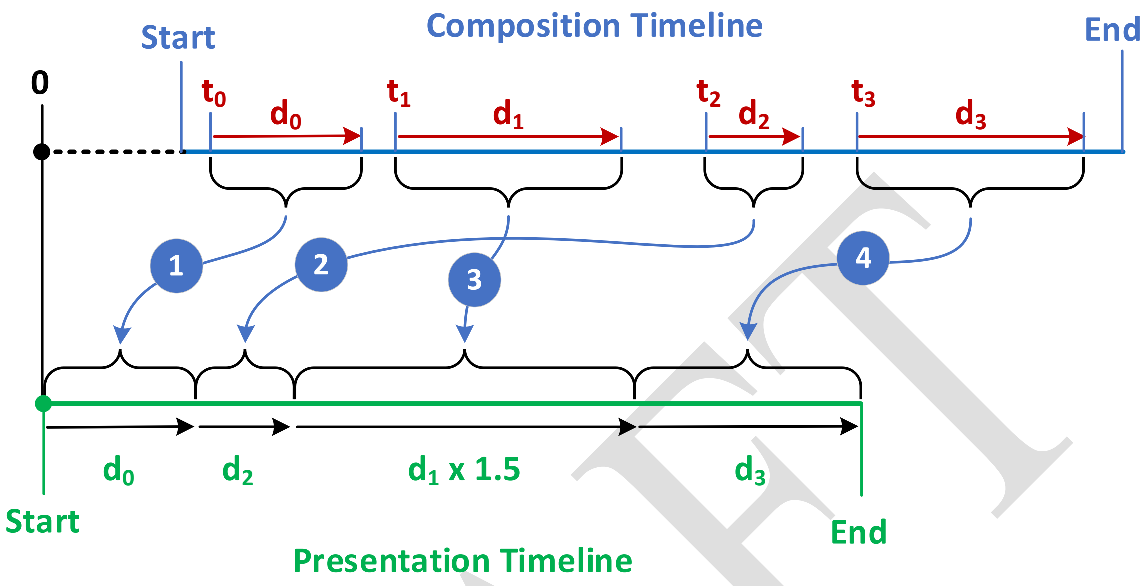


Figure 25: Edit List Example

The process for playout is to play the media according to the presentation timeline and map back to the composition timeline for what to playback when. The playout process steps through Table 16 in order and plays out the appropriate samples from the composition timeline.

Table 16: Presentation Timeline Playback

Edit index	Edit Tuple
1	$(t_0, d_0, 1.0)$
2	$(t_2, d_2, 1.0)$
3	$(t_1, d_1, 0.666)$
4	$(t_3, d_3, 1.0)$

There are some special values in the edit record providing certain functions, refer to the ISOBMFF-Core standard for more detail.

5.5.1.1.2 Sample Data Locations

Samples data locations are physical references to the sample data within the MediaDataBox; however, instead of separate physical references to each sample, the sample data locations reference to groups of samples, known as chunks. Chunks provide efficient storage by maintaining only references to chunks-not samples directly.

Figure 26 illustrates an example of chunk definition and how chucks map to a MediaDataBox. The figure shows a track with a list of samples ranging from Sample 1 to Sample 803 with the grouping of contiguous samples together into chunks. The chunks do not need to be the same size; for example, Chunk$_1$ contains three samples, and the remaining chunks all contain four samples. The figure shows when storing the chunk data within the MediaDataBox the chunks may be in anywhere within the MediaDataBox, e.g., Chunk$_2$ is towards the end of the MediaDataBox far away (offset-wise) from Chunk$_1$. To enable decoders to find the chunks, the encoder records the physical reference offset to each chunk in the ChunkOffsetBox.

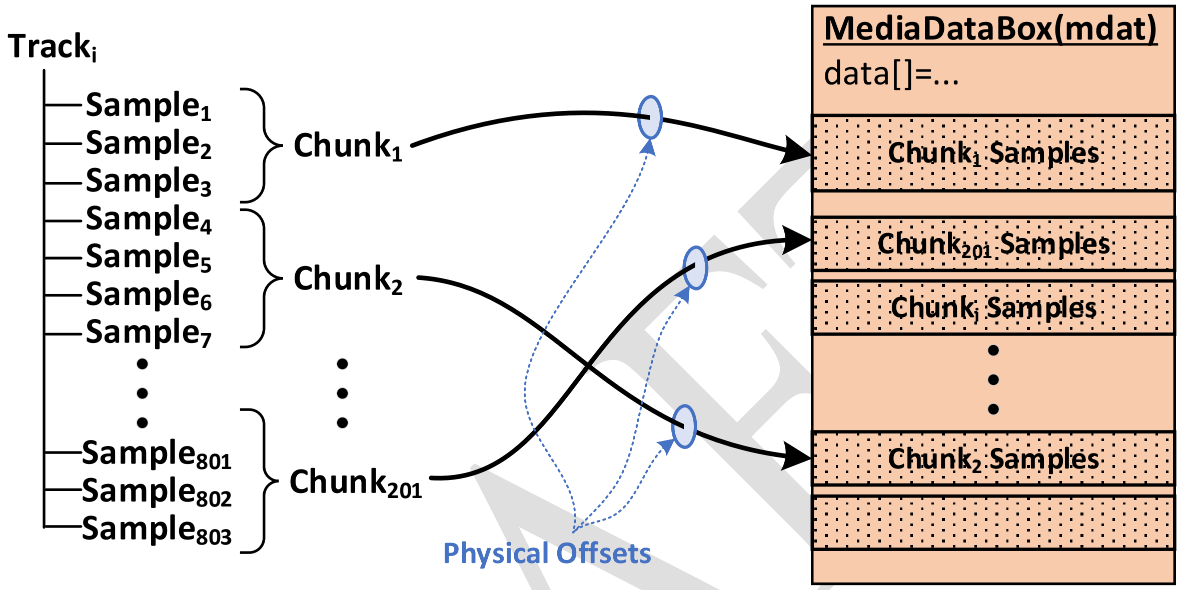


Figure 26: Chunks and Locations

Table 17 lists chunk offsets for the example in Figure 26. Each Chunk Offset is an offset from an origin which can be the start of the ISOBMFF-Core file, start of a box payload, or a different file containing the media. ISOBMFF-Core includes boxes within the track hierarchy for specifying what the origin is.

Table 17: Chunk Offset Example

Chunk Index	Chunk Offset
1	1000 + origin
2	100000 + origin
3	2000 + origin
…	…
201	1500 + origin

In addition to the chunks and their offsets, decoders need to map a sample to a chunk. ISOBMFF-Core uses a form of Run Length Encoding (RLE) to build a sample-to-chunk index. The pre-RLE sample-to-chunk index is a table mapping the sample number to the chunk number. Table 18 is a sample-to-chunk index for the Figure 26 example which shows each sample and the chunk to which it belongs. The green chunk is the only 3-sample chunk, while the orange and gray chunks are all 4-sample chunks. For a given sample number in the first row, the corresponding chunk number is in the second row.

Table 18: Sample-to-Chunk Example (Large Index)

Sample	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	S_{18}	S_{19}	…	S_{801}	S_{802}	S_{803}
Chunk	C_1	C_1	C_1	C_2	C_2	C_2	C_2	C_3	C_3	C_3	C_3	C_4	C_4	C_4	C_4	C_5	C_5	C_5	C_5	…	C_{201}	C_{201}	C_{201}

Table 19 shows an alternate form (i.e., Condensed Index) of Table 18 which only states the first sample for each chunk. Given the Sample number, a search of the top row determines which Chunk the sample is in on the second row. The third row in the table shows the Sample Count for each of the chunks, which is necessary for performing RLE.

Table 19: Sample-to-Chunk Example (Condensed Index)

Sample	S_1	S_4	S_8	S_{12}	S_{16}	…	S_{800}
Chunk	C_1	C_2	C_3	C_4	C_5	…	C_{201}
Sample Count	3	4	4	4	4		4

Recognizing chunks 2 through 201 all contain 4 samples per chunk, the RLE combines all those chunks together in the index. Additionally, since the sample numbers are sequentially in order (e.g., S_1, S_2, S_3, …) with the chunk's order and the count available for each group of chunks, the Sample row is redundant. Table 20 shows the RLE form of Table 19, showing the greater efficiency compared to Table 18. The SampleToChunkBox is a list of the RLE records.

Table 20: Sample-to-Chunk Example (RLE Index)

Chunk (C_i)	C_1	C_2
Sample Count (S_i)	3	4

Given a sample number, the RLE table, and the total number of chunks (known from other information in the ISOBMFF-Core file) the table's chunk number and sample count provide enough information to compute the chunk number for a given sample.

(Just did this to prove to myself it worked. We can cut this later.)

```
Find Chunk $C_y$ for Sample $S_x$
        ($T_C$=total number of chunks, $N_{RLE}$ = number of RLE records, e.g., 2 in the above table)
        Loop Starting with the second RLE record, $i = 2$
            if $i > N_{RLE}$, then $C_i = T_c$
            $N_{C_{i-1}} = C_i - C_{i-1}$            Number of chunks in the RLE $C_{i-1}$ group.
            $N_S = N_{C_{i-1}} \cdot S_{i-1}$            Number of samples in the $C_{i-1}$ group.
            If $S_x < N_S$ then
                $C_y = \lfloor S_x / S_{i-1} \rfloor + C_{i-1}$        $\lfloor \blacksquare \rfloor$ is the floor operation.
            End if
            $S_x = S_x - N_S$        Reduce the number of samples by the chunk count.
        End Loop
```

5.6 Media Data Details

There are two types of media data areas, non-identified and identified. Non-identified media data is a large block of bytes to store media data. For example, a block of the file that is 200 Mbytes in length as raw bytes.

Identified media data is the same as non-identified except there is an identifier preceding the data block. For example, a 4-byte value followed by a block of the file that is 200 Mbytes in length as raw bytes.

5.6.1 MediaDataBox

The MediaDataBox is a non-identified box containing one element, an array of bytes for storing audio, video, and other media types. The MediaDataBox does not have a formal or required organization of the media in the box. Other boxes such as the MovieBox reference the media using file-based physical references (i.e., origin of reference is the first byte of the file). Figure 27 illustrates a MediaDataBox example with the data array filled with two "Data Blocks" for Audio and Movie.

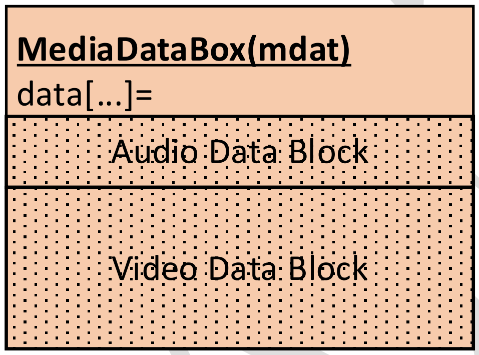


Figure 27: MediaDataBox Example

5.6.2 IdentifiedMediaDataBox

The IdentifiedMediaDataBox is an identified box containing two elements, an identifier and an array of bytes for storing audio, video, and other media types. The IdentifiedMediaDataBox does not have a formal or required organization of the media in the box. Other boxes such as the MovieBox reference the media using file-based physical references (i.e., origin of reference is the first byte of the file). Figure 28 illustrates a IdentifiedMediaDataBox example with the data array filled with two "Data Blocks" for Audio and Movie.

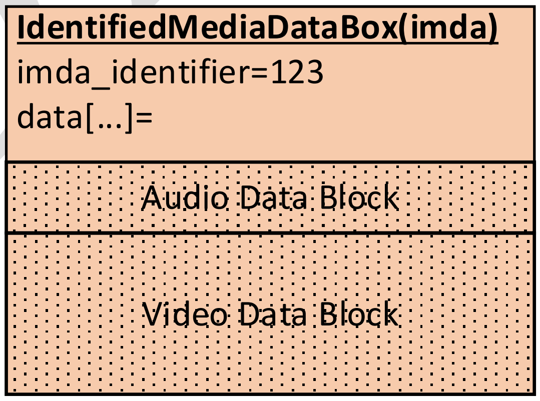


Figure 28: IdentifiedMediaDataBox Example

6 High Efficiency Image File Format (HEIF)

This section is reserved for an overview of HEIF.

Appendix A Syntactic Description Language (SDL)

ISO/IEC 14496-1 describes the employment of the Syntactic Description Language (SDL) for the bitstream syntax and rules used in ISO documents. SDL lends itself to object-oriented data representations and assumes an object-oriented framework where bitstream units consist of classes. SDL extends the typing system of the C++ and Java programming languages by providing means to define bitstream-level quantities and their parsing.

While SDL defines four elementary data types, namely, constant-length direct representation bit fields, variable-length direct representation bit fields, constant-length indirect representation bit fields, and variable-length indirect representation bit fields, the focus here is on the constant-length direct representation as it applies to GIMI.

In addition, Classes are the means for defining composite types or objects. Syntactic Flow Control provides constructs for conditional and repetitive parsing. This appendix reviews these topics as applied to GIMI.

A.1 Constant-Length Direct Representation Bit Fields

Rule E.1 Elementary Data Types: ISO/IEC 14996-1 requires the following for representation of constant-length direct representation bit fields as:

[**aligned**] **type**[(*length*)] element_name [= value]; // C++-style comments allowed

The **type** allowed includes: **int** for signed integer, **unsigned int** for unsigned integer, **double** for floating point, and **bit** for raw binary data. The *length* is the length of the element in bits as stored in the bitstream. The length of a double can be either 32 or 64 bit only. The value can only be present when fixed and may include a range such as ‘0x00..0x7F’. The type and optional length are always present in a parsable bitstream. The keyword **aligned** means the alignment of data is on a byte boundary; however, the values 8, 16, 32, 64 and 128 signal alignment on other than a byte boundary. Skipped bits are set to zero (0). Data is most significant bit first, and most significant byte first.

The keyword **const** defines a constant, for example:

const int MY_VALUE=127; //non-parsable constant

The prefix **0b** designates binary values, while the prefix **0x** designates hexadecimal values. An optional period placed after every four digits can help readability. Thus, 0x0C is equivalent to 0b0000.1100.

Rule E.2: Look-ahead parsing: This feature permits examining the following bits in a bitstream without consuming these bits. Placing the character ‘*’ after the length parentheses achieves this behavior:

[**aligned**] **type** (length) * element_name;

For example, performing a check on the value of the next 16 bits in the bitstream prior to advancing the current position in the bitstream. As an example:

aligned unsigned int (16) * next_bytes;

A.2 Composite Data Types

Classes form the basis for creating objects, i.e., boxes in ISOBMFF. Although SDL defines rules for abstract classes, expandable classes, parameter type classes, partial and implicit arrays, this section limits SDL to the topic of classes used in GIMI. The class definition is:

Rule C.1: Classes

[**aligned**] [abstract] [expandable[(maxClassSize)]] **class** object_name [**extends** parent_class]
[: bit(length) [id_name=] object_id | id_range | extended_id_range] {
[element; …] // zero or more elements
}

The optional word **extends** indicates that the **class** derives from another class. Thus, all information within the base class is also present in the derived class. As such, all base class information precedes in the bitstream any additional declarations specified in the new class.

The form for Rule C.1 used throughout the GIMI standard is:

[**aligned**] **class** object_name [**extends** parent_class]
{
[element; …] // zero or more elements
}

The elements within the curly brackets are he definitions of either bitstream components (discussed in A1), or control flow elements discussed below.

A.3 Arithmetic and Logical Expression

SDL allows all standard arithmetic and logical operators of C++, including their precedence rules.

A.4 Syntactic Flow Control

Syntactic Flow Control provides constructs for conditional parsing and repetitive parsing. Rule FC.1 is common in ISOBMFF. SDL supports other flow control constructs such as 'for', 'while', and 'do' loops as well.

Rule FC.1: Flow Control Using If-Then-Else

```
if (condition) {
    …
} [else if (condition) {
   …
} [else {
   …
}]
```