ISO/IEC JTC 1/SC 29/WG 03 N1515

**ISO/IEC JTC 1/SC 29/WG 03  
MPEG Systems   
Convenorship: KATS (Korea, Republic of)**

**Document type:** Output Document

**Title:** Exploration on application specific metadata definition and carriage in media AUs

**Status:** Approved

**Date of document:** 2025-01-24

**Source:** ISO/IEC JTC 1/SC 29/WG 03

**No. of pages:** 31 (with cover page)

**Email of Convenor:** young.L @ samsung . com

**Committee URL:** <https://isotc.iso.org/livelink/livelink/open/jtc1sc29wg3>

**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 03 MPEG SYSTEMS**

**ISO/IEC JTC 1/SC 29/WG 03 N1515**

**March 2025, online**

|  |  |
| --- | --- |
| **Title** | **Exploration on application specific metadata definition and carriage in media AUs** |
| **Source** | **WG 03, MPEG Systems** |
| **Status** | **Approved** |
| **Serial Number** | **25051** |

# Scope

Various media applications make use of auxiliary data, sometimes metadata, accompanying media streams. For example, network-rendered media for immersive media distribution are getting recently popular, including cloud gaming, XR services, and metaverse related applications. In this context, the definition of and carriage of sample-auxiliary metadata (i.e. metadata that is to be processed synchronously with the media sample) for network rendered media is an important aspect. The metadata should be “close” to the media sample to be used in the final device rendering process.

This exploration provides background information on various type of applications using auxiliary data per media sample such as network-based and split-rendering and analyses ideas around definition and carriage and metadata and discusses some open issues for further study.

In particular it invites for input for MPEG#148 on

* Use cases:
  + additional use cases on the necessity of media-type independent sample metadata
  + architectures and call flows supporting the use cases to understand the involved issues when delivering the data over different networks
* Design principles:
  + What format for representing metadata in an access unit?
  + Is the metadata by itself an access unit or is it part of an access unit?
  + More generally, can an elementary stream constitute of a sequence of access units each carrying metadata?
* Solution:
  + draft requirements on metadata and the carriage of the metadata

**Table of Contents**

[Scope 1](#_Toc194682470)

[1 Introduction 3](#_Toc194682471)

[2 Scope of exploration 3](#_Toc194682472)

[3 Use Case 1: Split Rendering and OpenXR 4](#_Toc194682473)

[3.1 Description 4](#_Toc194682474)

[3.2 Considered Metadata 9](#_Toc194682475)

[3.3 Rendering system of immersive content for various types of devices 10](#_Toc194682476)

[3.3.1 Server-side rendering system 10](#_Toc194682477)

[3.3.2 Renderer and player for presenting immersive content 10](#_Toc194682478)

[3.3.3 Renderer and authoring tool for creating immersive content 12](#_Toc194682479)

[3.3.4 Rendering server on AWS 14](#_Toc194682480)

[3.3.5 Considered Metadata and its carriage format 14](#_Toc194682481)

[4 Potential Solution for Carriage of Metadata 15](#_Toc194682482)

[4.1 Introduction 15](#_Toc194682483)

[4.2 SEI Message-based Carriage 15](#_Toc194682484)

[4.3 RTP Header Extension based on 3GPP draft solution 16](#_Toc194682485)

[4.4 ITU-T T35 message 16](#_Toc194682486)

[4.5 MMT Signaling message 16](#_Toc194682487)

[4.6 Signalling in stream of access units 16](#_Toc194682488)

[4.7 Analysis of metadata carriage in MPEG-4 part 3 18](#_Toc194682489)

[4.7.1 Background 18](#_Toc194682490)

[4.7.1 Feedback from Audio experts at MPEG #150 22](#_Toc194682491)

[4.7.2 Possible approaches for metadata carriage in AAC access unit 23](#_Toc194682492)

[5 Evaluation of different solutions 28](#_Toc194682493)

[5.1 Issues to study 28](#_Toc194682494)

[5.2 Metadata Semantics 28](#_Toc194682495)

[5.3 Metadata Definition 29](#_Toc194682496)

[5.4 Metadata Carriage 29](#_Toc194682497)

[6 Potential Work in MPEG 29](#_Toc194682498)

[6.1 General 29](#_Toc194682499)

[6.2 Possible next steps 30](#_Toc194682500)

[7 References 30](#_Toc194682501)

# 1 Introduction

Generally, the metadata definition and carriage of sample-auxiliary metadata (i.e. metadata that is to be processed synchronously with the media sample) for media applications is an important aspect.

The metadata should be “close” to the media sample to be used in the final device rendering process.

This exploration provides background information on various type of applications using auxiliary data per media sample such as network-based and split-rendering and analyses ideas around definition and carriage and metadata and discusses some open issues for further study.

In particular it invites for input for MPEG#148 on:

* Use cases:
  + additional use cases on the necessity of media-type independent sample metadata
  + architectures and call flows supporting the use cases to understand the involved issues when delivering the data over different networks
* Design principles:
  + What format for representing metadata in an access unit?
  + Is the metadata by itself an access unit or is it part of an access unit?
  + More generally, can an elementary stream constitute of a sequence of access units each carrying metadata?
* Solution:
  + draft requirements on metadata and the carriage of the metadata

# 2 Scope of exploration

Figure 1 depicts the application framework in which the exploration takes place. That is an encoder produces a media elementary streams which is enriched with access unit metadata. Upon reception by the client, an API is responsible of extracting the inserted metadata and passing it over to the application by maintaining the association between the metadata and the decoded media data.

A diagram of a variety of squares

AI-generated content may be incorrect.

Figure 1 - Exploration scope

As illustrated in Figure 1, its is important to note that:

**The exploration assumes that a player has an API in front of the media decoders (audio and video) in order to extract the metadata contained in the access units.**

**If a decoder does parse a bitstream with such metadata, the exploration considers this is an abnormal case and it is nice-to-have if the decoder does not crash.**

# 3 Use Case 1: Split Rendering and OpenXR

## 3.1 Description

Users are always looking for more realistic and high-fidelity immersive experiences in gaming, entertainment, and communication applications and services. At the same time, more and more users are relying on mobile and portable devices and HMDs for consuming these services. The development of the Metaverse is expected to accelerate these trends and culminate the emergence of advanced and lightweight glasses and HMDs.

These two concurrent trends result in challenges for managing the processing power and battery life on these devices. Immersive high-fidelity experiences require immense graphics processing resources that come with high power consumption, which cannot be reconciliated with the capabilities and design goals of the XR devices/glasses.

Split Rendering is an interesting technology that provides new requirements for encoders, decoders and systems technologies. To meet high user quality, the "roundtrip latency", referred to as motion-to-render-to-photon latency (M2R2P) is required to be in the range of at most 100ms, preferably less than 60ms. In addition, new type of metadata data needs to be exchanged and this metadata exchange also needs to meet the latency requirements.

Split rendering has been identified as a promising approach to address these challenges. With split rendering, the whole rendering process or parts thereof are performed in the network, for example in an edge that is supported by a reliable and optimized network such as 5G.

A basic architecture for Split Rendering is shown in Figure 1. In this case an application resides in a network server that runs a game engine to generate and render complex scenes. For a specific user, pose, controller and tracking information is used in order to render the scene. In addition, a specific viewport of the scene is rendered by the user and regular encoders for audio and video send regular media data to a device. The device decodes the information, and finally send this to a composition process, that does the final presentation using the latest pose and environment information to the users, for example by Asynchronous Time Warping (ATW), etc.

Note that to meet high user quality, the "roundtrip latency", referred to as motion-to-render-to-photon latency (M2R2P) is required to be in the range of at most 100ms, preferably less than 60ms. This provides challenges for systems, but if fulfilled, offers many new application services.



Figure 2 Basic Split rendering architecture

One configuration of split rendering is the so-called *Pixel Streaming*. In Pixel Streaming, the network server receives the configuration of the XR session on the device, renders (off-screen) the audio and video of the 3D scene, and streams the rendered media on the downlink to the device. The device can use OpenXR or a similar XR runtime system to display/render the pre-rendered media.

The following call flow shows the operation of split rendering as defined in 3GPP TS26.565:

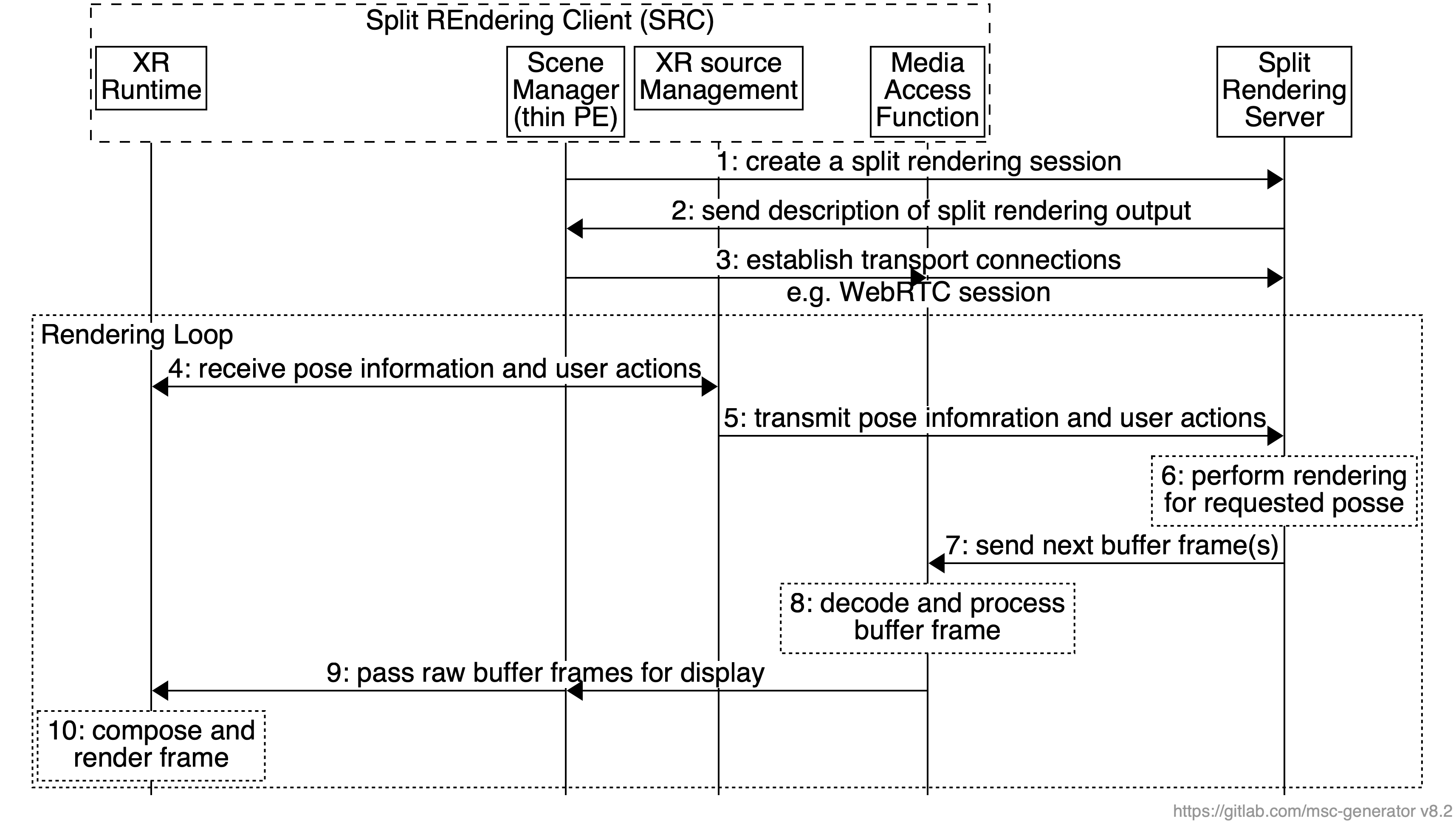


Figure 3 Call flow shows the operation of split rendering as defined in 3GPP TS26.565

As an example for a device runtime API for XR, OpenXR is an API that is developed by the Khronos Group for developing XR applications that address a wide range of XR devices. XR refers to a mix of real and virtual world environments that are generated by computers through interactions by humans. XR includes technologies such as virtual reality (VR), augmented reality (AR) and mixed reality (MR). OpenXR is the interface between an application and XR runtime. The runtime handles functionality such as frame composition, user-triggered actions, and tracking information.

OpenXR is designed to be a layered API, which means that a user or application may insert API layers between the application and the runtime implementation. These API layers provide additional functionality by intercepting OpenXR functions from the layer above and then performing different operations than would otherwise be performed without the layer. In the simplest cases, the layer simply calls the next layer down with the same arguments, but a more complex layer may implement API functionality that is not present in the layers or runtime below it. This mechanism is essentially an architected "function shimming" or "intercept" feature that is designed into OpenXR and meant to replace more informal methods of "hooking" API calls.

Applications may determine the API layers that are available to them by calling the [xrEnumerateApiLayerProperties](https://microsoft.github.io/OpenXR-MixedReality/openxr_preview/specs/openxr.html#xrEnumerateApiLayerProperties) function to obtain a list of available API layers. Applications then may select the desired API layers from this list and provide them to the [xrCreateInstance](https://microsoft.github.io/OpenXR-MixedReality/openxr_preview/specs/openxr.html" \l "xrCreateInstance) function when creating an instance.

API layers may implement OpenXR functions that may or may not be supported by the underlying runtime. In order to expose these new features, the API layer must expose this functionality in the form of an OpenXR [extension](https://microsoft.github.io/OpenXR-MixedReality/openxr_preview/specs/openxr.html#extensions). It must not expose new OpenXR functions without an associated extension.

An OpenXR instance is an object that allows an OpenXR application to communicate with an OpenXR runtime. The application accomplishes this communication by calling [xrCreateInstance](https://microsoft.github.io/OpenXR-MixedReality/openxr_preview/specs/openxr.html" \l "xrCreateInstance) and receiving a handle to the resulting [XrInstance](https://microsoft.github.io/OpenXR-MixedReality/openxr_preview/specs/openxr.html#XrInstance) object.

The [XrInstance](https://microsoft.github.io/OpenXR-MixedReality/openxr_preview/specs/openxr.html" \l "XrInstance) object stores and tracks OpenXR-related application state, without storing any such state in the application’s global address space. This allows the application to create multiple instances as well as safely encapsulate the application’s OpenXR state since this object is opaque to the application. OpenXR runtimes may limit the number of simultaneous [XrInstance](https://microsoft.github.io/OpenXR-MixedReality/openxr_preview/specs/openxr.html" \l "XrInstance) objects that may be created and used, but they must support the creation and usage of at least one [XrInstance](https://microsoft.github.io/OpenXR-MixedReality/openxr_preview/specs/openxr.html" \l "XrInstance) object per process.

Spaces are represented by [XrSpace](https://www.khronos.org/registry/OpenXR/specs/1.0/man/html/openxr.html" \l "XrSpace) handles, which the application creates and then uses in API calls. Whenever an application calls a function that returns coordinates, it provides an [XrSpace](https://www.khronos.org/registry/OpenXR/specs/1.0/man/html/openxr.html" \l "XrSpace) to specify the frame of reference in which those coordinates will be expressed. Similarly, when providing coordinates to a function, the application specifies which [XrSpace](https://www.khronos.org/registry/OpenXR/specs/1.0/man/html/openxr.html" \l "XrSpace) the runtime to be used to interpret those coordinates.

OpenXR defines a set of well-known reference spaces that applications use to bootstrap their spatial reasoning. These reference spaces are: VIEW, LOCAL and STAGE. Each reference space has a well-defined meaning, which establishes where its origin is positioned and how its axes are oriented.

Runtimes whose tracking systems improve their understanding of the world over time may track spaces independently. For example, even though a LOCAL space and a STAGE space each map their origin to a static position in the world, a runtime with an inside-out tracking system may introduce slight adjustments to the origin of each space on a continuous basis to keep each origin in place.

Beyond the well-known reference spaces, runtimes expose other independently tracked spaces, such as a pose action space that tracks the pose of a motion controller over time.

The following figure depicts the lifecycle of an application that uses OpenXR for interaction and rendering with/to an HMD.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 4 OpenXR application lifecycle

After creating an OpenXR session, the application starts a frame loop. The frame loop is executed for every frame. The frame loop consists of the following steps:

1. Synchronize actions: this step consists of retrieving the action state, e.g. the status of the controller buttons and the associated pose. During this step, the application also establishes the location of different trackables. The application may also send haptics feedback.
2. Start a new frame: this step starts with waiting for a frame to be provided by the XR runtime. This step is necessary to synchronize the application frame submission with the display. The xrWaitFrame function returns a frame state for the requested frame that includes a predictedDisplayTime, which is a prediction of when the corresponding composited frame will be displayed. This information is used by the application to request the predicted pose at display. Once the xrWaitFrame function completes, the application calls xrBeginFrame to signal the start of the rendering process.
3. Retrieve rendering resources: the application starts by locating the views in space and time by calling the xrLocateViews function, provided with the predicted display time and the XR space. It then acquires the swap chain image associated with every view of the composition layer. It waits for the swap chain image to be made available so it can write into it.
4. Rendering: the application then performs its rendering work. This is for instance what the scene manager is tasked with. It iterates over the scene graph nodes and renders each object to the view. This step usually uses a Graphics Framework such Vulkan, OpenGL, or Direct3D to perform the actual graphics operations.
5. Release resources: once the rendering is done for a view, the application releases the corresponding swap chain image. Once all views are rendered, it sends them for display by calling the xrEndFrame function.

In terms of rendering operation, the relevant part is located between the call to xrBeginFrame and the call to xrEndFrame on the bottom right part of the diagram.

When the application calls the xrEndFrame function, the application provides the structure XrFrameEndInfo which contains all necessary information to render the frame that is:

* The time at which this frame should be displayed.
* The mode to be used for blending the user’s environment with the submitted frame.
* One or more layers which compose the submitted frame, where each composition layer provides the XR space, pose, fov, and the corresponding swapchain image(s).

A key feature of the XR runtime is its ability to perform layer composition. A Compositor in the runtime is responsible for taking all the received layers from xrEndFrame calls, performing any necessary corrections such as pose correction and lens distortion, compositing them, and then sending the final frame to the display. An application may use multiple composition layers for its rendering. The number of supported composition layers may be queried by the application.

OpenXR supports different types of layers, with the main ones being:

* Projection Composition Layer: represents planar projected images, one rendered for each eye using a perspective projection.
* Quad Composition Layer: is useful for rendering user interface elements or 2D content on a planar area in the world.
* Cube Composition Layer: consists of a cube map with 6 views to be rendered by the application.
* Equirectangular Composition Layer: consists of an equirectangular image that is mapped onto the inside of a sphere in the world.
* Depth Composition Layer: provides an extra composition layer to allow applications to submit depth maps to assist with the pose correction of projected images of a project layer.

The next figure depicts an example of a projection composition layer and the resulting composited distorted image (image courtesy of Khronos).

A screenshot of a video game

Description automatically generated

Figure 2 – Example illustrating composition of a stereoscopic image submitted to the Compositor

Another relevant configuration when setting up the XR session is the choice of the view configuration, which depends on the target device and its capabilities. Mono and Stereo are natively supported by all XR runtimes. Some advanced types like the primary quad, defined as a vendor extension provide support for foveated rendering.

## 3.2 Considered Metadata

As discussed in section 2, the XR runtime expects each rendered frame to be accompanied by a description of the *pose* that was used to render that frame. Other information such as the FoV and the XR space may be static and do not need to be sent with every frame. The XR runtime uses the pose information to perform any pose correction prior to display.

It can also be assumed that the audio renderer will perform similar pose correction prior to playing back the audio frame. Pose correction is essential for split rendering as the round-trip time from pose acquisition to displaying the rendered media on the device may be significant, given that the rendering happens in the network.

In addition to the pose, the Split Rendering Server may also provide a list of the actions that have been processed prior to the network rendering operation for a specific frame. A possible Metadata is considered here:

* **xrpi\_actions\_present:** indicates if a list of actions is present.
* **xrpi\_timestamp:** the wallclock timestamp of the render pose.
* **xrpi\_x, xrpi\_y, xrpi\_z:** the coordinates of the position of the render pose.
* **xrpi\_rx, xrpi\_ry, xrpi\_rz, xrpi\_rw:** the components of the quaternion for the rotation of the render pose.
* **xrpi\_action\_count:** the number of actions that are processed prior to rendering with the current render pose.
* **xrpi\_action\_id:** an identifier of the action that was processed prior to rendering with the current render pose.

Such metadata may be applicable to both, video and audio.

Note that uplink metadata is not considered in the contribution.

## 3.3 Rendering system of immersive content for various types of devices

### 3.3.1 Server-side rendering system

NHK has developed the system shown in Figure 3 to use immersive content on various types of devices such as head-mounted displays (HMD) and tablets.

**A diagram of a video game

Description automatically generated**

Figure 3 Overview of presentation of immersive content on various types of devices

Immersive content contains information on a 3D space. It is constructed by arranging video objects such as 360-degree video, volumetric video, 2D video, and so on in a 3D space by utilizing scene descriptions. The content is not intended for devices with specific display capabilities.

From 3D space, the renderer generates the 2D video to be presented on a user’s device depending on the device capability, such as its display resolution, field of view (FoV), and frame rate, in addition to the user’s viewpoint position and direction. This mechanism makes it possible to uniformly provide immersive content to various types of devices with different display capabilities [2].

### 3.3.2 Renderer and player for presenting immersive content

Figure 4 illustrates the functional blocks of a system that renders on a server and presents immersive content on a player on a device.

**A black screen with white text

Description automatically generated**

Figure 4 Functional blocks of the rendering server and player

The developed system uses volumetric video objects including point cloud format (Video-based Point Cloud Compression (V-PCC) format) and dynamic mesh format (Video-based Dynamic Mesh Coding (V-DMC) format) as well as 360-degree video, on the premise that decoding and rendering are performed in real time.

Scene descriptions, 360-degree video objects, and volumetric video objects are all divided at fixed intervals and sent from the transmitter (the left side in Fig. 4) to the rendering server (center of Fig.4) in real time by utilizing MPEG Media Transport (MMT).

The rendering server stores these received signals. It parses the scene description of the requested content when there is a content request from the player. Figure 3 shows information to be transferred between the rendering server and player.

1. When a player requests content, it notifies the rendering server of its device capability including its display resolution, FoV, and frame rate.
2. The rendering server parses the scene description, identifies video objects needed to construct a 3D space, decodes the necessary streams, and then places them in the 3D space.
3. The player constantly notifies the rendering server of the user’s viewpoint position and direction on the basis of HMD sensor or user operation.
4. The rendering server generates 2D video from the 3D space in accordance with the users’ viewpoint position and direction.
5. The generated 2D video is sent to the player by either MMTP/UDP or MMTP/HTTP together with the virtual camera position and direction at the time of generating 2D video.
6. The player presents the image extracted from the generated 2D video that matches the viewpoint position and direction at the time of presentation.

A black background with a blue dotted line

Description automatically generated

Figure 5 Information to be transferred between rendering server and player

For presentation devices without reported device capability, a 2D video is generated on the basis of the recommended viewport information included in the scene description and sent in one direction. In this case, users cannot change their viewpoint position or direction.

Figure 6 shows the screen of a player using the rendering system: HMD (left side), tablet (center), and TV set (right side).

**A screen with a wire attached to it

Description automatically generated**

Figure 6 Developed player on HMD, tablet, and TV set.

### 3.3.3 Renderer and authoring tool for creating immersive content

To popularize immersive content, a tool for easily creating immersive content is necessary. NHK has developed an authoring tool that uses the abovementioned rendering server. Figure 7 illustrates the functional blocks of the developed authoring tool and rendering server.

**A computer screen shot of a video game

Description automatically generated**

Figure 7 Functional blocks of the rendering server and authoring tool

The authoring tool connects to the user’s selected rendering server and notifies the server to start the authoring operation.

When an authoring tool is connected, the following behavior of the rendering server differs from the behavior when a player is connected.

* The authoring tool becomes the source of scene descriptions and video objects.
* The rendering server receives properties such as the position, orientation, and size of video objects in accordance with user operation.
* The rendering server can randomly access video objects by caching the decoding results of video objects.
* The rendering server simultaneously generates multiple 2D videos in accordance with viewpoint position and direction and FoVs that are set separately. On the authoring tool, one of the images can be presented on an HMD.

Figure 6 shows an example of displaying multiple 2D videos on the authoring tool.

**A person using a computer

Description automatically generated**

Figure 8 Developed authoring tool

The authoring tool has multiple views at the same moment and allows users to change their viewpoint position and direction and their FoV. Therefore, immersive content can be created by arranging video objects while checking how they look from various positions and directions in the 3D space. Content can also be presented on an HMD, and in this case, the image is presented in accordance with the movement of the HMD.

The 2D videos presented on the authoring tool are all generated by the rendering server. The operations for generating the 2D videos are the same as those for players described in Section 2.2.

The authoring tool and players were exhibited at NHK STRL open house in June 2023 [3].

### 3.3.4 Rendering server on AWS

The developed rendering server can be operated on Amazon Elastic Compute Cloud (AWS EC2). We are conducting verification in a limited environment between AWS EC2 and NHK STRL. We plan to conduct verification in a more general network environment in the future.

### 3.3.5 Considered Metadata and its carriage format

The following metadata is used in the developed system.

1. Device capability

|  |  |
| --- | --- |
| When: | At the beginning of the session |
| Direction: | Player/authoring tool -> server |
| Format: | RTSP |
| Contained information: | Display resolution, FoV, and presentation frame rate |

1. Viewport position and direction

|  |  |
| --- | --- |
| When: | During presentation/creation |
| Direction: | Player/authoring tool -> server |
| Format: | MMT signalling message |
| Contained information: | User’s viewport position and direction (**viewpoint\_pos\_x**, **viewpoint\_pos\_y**, **viewpoint\_pos\_z**,  **viewpoint\_yaw**, **viewpoint\_pitch**, **viewpoint\_roll**) |

1. Rendered position and direction

|  |  |
| --- | --- |
| When: | During presentation/creation |
| Direction: | Server -> player/authoring tool |
| Format: | MMT signalling message (identical format with item 2) |
| Contained information: | * Camera position and direction at generating 2D video (**viewpoint\_pos\_x**, **viewpoint\_pos\_y**, **viewpoint\_pos\_z**,  **viewpoint\_yaw**, **viewpoint\_pitch**, **viewpoint\_roll**) * MPU sequence number for synchronization with the corresponding 2D video |

1. User operation for arranging video objects

|  |  |
| --- | --- |
| When: | During creation |
| Direction: | Authoring tool -> server |
| Format: | MMT signalling message |
| Contained information: | Video object index, Translation/Rotation/Scale of video object,  and frame number being edited for user operation |

In addition to the abovementioned metadata, for random access, WebAPIs are implemented to notify the rendering server of the frame number to be presented.

# 4 Potential Solution for Carriage of Metadata

## 4.1 Introduction

Different ways of carrying such split rendering metadata. Examples include RTP payload headers, or other system level information. However, the information is closely attached to the rendered original source content, so attaching it to media samples inband is another suitable option.

## 4.2 SEI Message-based Carriage

As an example, an in-band carriage as SEI message is provided in Table 1 making use of the information in provided in clause 3.

Table 1 Potential SEI message for in-band carriage of render pose information

|  |  |
| --- | --- |
| **xr\_render\_pose\_info(payloadSize)** { | Descriptor |
| xrpi\_actions\_present | u(1) |
| xrpi\_reserved | u(7) |
| xrpi\_timestamp | u(64) |
| xrpi\_x | f(32) |
| xrpi\_y | f(32) |
| xrpi\_z | f(32) |
| xrpi\_rx | f(32) |
| xrpi\_ry | f(32) |
| xrpi\_rz | f(32) |
| xrpi\_rw | f(32) |
| if (xrpi\_actions\_present) { |  |
| actions\_count | u(8) |
| for (i=0;i<xrpi\_actions\_present;i++) { |  |
| xrpi\_action\_id | u(16) |
| } |  |
| } |  |
| } |  |

## 4.3 RTP Header Extension based on 3GPP draft solution

Split Rendering (SR) Media Service Enabler (MSE) is being defined in 3GPP SA4 in specification TS26.565. The SR MSE currently relies on a solution for the carriage of the rendered pose that is based on an RTP header extension defined in a permanent document for the 5G\_RTP work item ([S4-231468](https://www.3gpp.org/ftp/TSG_SA/WG4_CODEC/TSGS4_125_Gotheneburg/Docs/S4-231468.zip) section 4.3).

The RTP header extension is currently defined as follows:

0 1 2 3  
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1  
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
| 0x100 | appbits| length |  
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
| ID | L=36+2n | x …

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| y …

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | z …

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | rx …

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | ry …

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | rz …

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | rw …

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
| | timestamp …

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| timestamp continued …

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| timestamp continued | action\_id #1 |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| action\_id #2 | ... |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

All fields of this header extension reflect values that are retrieved directly from the XR Runtime and received by the split rendering server or interpolations/extrapolations thereof.

Split Rendering Client and Server start by establishing a configuration of the split rendering session on the server side that aligns with the XR session on the client side. These fields are highly dependent on the XR Runtime that is used on the client side, e.g. the semantics and units may be different from XR Runtime to another. 3GPP SA4 uses OpenXR as a reference for XR Runtimes and their APIs.

## 4.4 ITU-T T35 message

https://www.itu.int/en/ITU-T/inr/forms/Pages/t35.aspx

## MMT Signaling message

See clause 3.5.

## Signalling in stream of access units

In MPEG-4 part 1 Systems, the concept of elementary stream is defined as “consecutive flow of mono-media data from a single source entity to a single destination entity on the compression layer”. Furthermore, elementary streams are made of access units which are the “smallest individually accessible portion of data within an elementary stream to which unique timing information can be attributed”.

Deviating a bit from the formal definition, MPEG-4 part 1 further defines sub variants of elementary streams to carry metadata. For instance, the Object Content Information (OCI) stream “an elementary stream that conveys time-varying object content information, termed OCI events. Each OCI event consists of a number of OCI descriptors.”

Table 2 - OCI events definition

|  |  |
| --- | --- |
| **OCI\_Events syntax** | **OCI\_Events semantics** |
| aligned(8) expandable(228-1) class OCI\_Event {  bit(15) eventID;  bit(1) absoluteTimeFlag;  bit(32) startingTime;  bit(32) duration;  OCI\_Descriptor OCI\_Descr[1 .. 255];  } | eventID – contains the identification number of the described event that is unique within the scope of this OCI stream.  absoluteTimeFlag – indicates the time base for startingTime as described below.  startingTime – indicates the starting time of the event in hours, minutes, seconds and hundredth of seconds. The format is 8 digits, the first 6 digits expressing hours, minutes and seconds with 4 bits each in binary coded decimal and the last two expressing hundredth of seconds in hexadecimal using 8 bits.  EXAMPLE ⎯ 02:36:45:89 is coded as “0x023645” concatenated with “0b0101.1001” (89 in binary), resulting to “0x02364559”.  If absoluteTimeFlag is set to zero, startingTime is relative to the object time base of the corresponding object. In that case it is the responsibility of the application to ensure that this object time base is conveyed such that startingTime can be identified unambiguously (see **Error! Reference source not found.**). If absoluteTimeFlag is set to one, startingTime is expressed as an absolute value, refering to wall clock time.  duration – contains the duration of the corresponding object in hours, minutes, seconds and hundredth of seconds. The format is 8 digits, the first 6 digits expressing hours, minutes and seconds with 4 bits each in binary coded decimal and the last two expressing hundredth of seconds in hexadecimal using 8 bits.  OCI\_Descr[] – an array of one up to 255 OCI\_Descriptor classes as specified in **Error! Reference source not found.**. |

An example of such metadata is the SmpteCameraPosition also defined in MPEG-4 part 1.

Table 3 - SmpteCameraPosition OCI descriptor

|  |
| --- |
| class SmpteCameraPositionDescriptor extends OCI\_Descriptor : bit (8) tag=SmpteCameraPositionDescrTag {  unsigned int (8) cameraID;  unsigned int (8) parameterCount;  for (i=0; i<parameterCount; i++) {  bit (8) parameterID;  bit (32) parameter;  }  } |

Therefore, MPEG-4 part 1 System does define derivative type of elementary stream to carry metadata and consider those streams to inherit from the elementary stream concept.

Also to be noted is that the RFC 3640 defines the carriage of MPEG-4 part 1 elementary streams over RTP **Error! Reference source not found.**. The benefit of this approach is that MPEG-4 part 1 defines many types of metadata that are carried in a generic way and thus allows the RFC 3640 to transport any MPEG-4 part 1 metadata in a agnostic manner.

From RFC 3640, 2.11. Global Structure of Payload Format:

The RTP payload following the RTP header, contains three octet aligned data sections, of which the first two MAY be empty, see Figure 1.

+---------+-----------+-----------+---------------+

| RTP | AU Header | Auxiliary | Access Unit |

| Header | Section | Section | Data Section |

+---------+-----------+-----------+---------------+

<----------RTP Packet Payload----------->

Figure 1: Data sections within an RTP packet

## Analysis of metadata carriage in MPEG-4 part 3

### 4.7.1 Background

In ISO/IEC 14496-3, standard compiles an overview of tools for audio storage and transmission. The following table 0.1 is extracted from it.

Table 0.4 – MPEG-4 Audio multiplex, storage and transmission formats

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Format | Functionality defined in MPEG-4: | Functionality originally defined in: | Description |
| Multiplex | M4Mux | ISO/IEC 14496-1  (normative) | - | MPEG-4 Multiplex scheme |
| LATM | ISO/IEC 14496-3  (normative) | - | Low Overhead Audio Transport Multiplex |
| Storage | ADIF | ISO/IEC 14496-3  (informative) | ISO/IEC 13818-7  (normative) | Audio Data Interchange Format,  (AAC only) |
| MP4FF | ISO/IEC 14496-12  (normative) | - | MPEG-4 File Format |
| Transmission | ADTS | ISO/IEC 14496-3  (informative) | ISO/IEC 13818-7  (normative, exemplarily) | Audio Data Transport Stream,  (AAC only) |
| LOAS | ISO/IEC 14496-3  (normative, exemplarily) | - | Low Overhead Audio Stream, based on LATM, three versions are available:  AudioSyncStream()  EPAudioSyncStream()  AudioPointerStream() |

**AAC raw data stream**

One top level payload (raw\_data\_block()) is mapped into one access unit. Subsequent access units form one elementary stream.

Table 1.A.5 – Syntax of raw\_data\_stream()

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| raw\_data\_stream() |  |  |
| { |  |  |
| while (data\_available()) { |  |  |
| raw\_data\_block(); |  |  |
| } |  |  |
| } |  |  |

**AAC raw data block**

The raw data block is a generic data structure of an AAC stream. A raw data block carries identifier field which indicates the type of the block.

Table 4.6 – Syntax of top level payload for audio object types AAC Main, SSR, LC, and LTP (raw\_data\_block())

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| raw\_data\_block() |  |  |
| { |  |  |
| while( (id = **id\_syn\_ele**) != ID\_END ){ | **3** | **uimsbf** |
| switch (id) { |  |  |
| case ID\_SCE: single\_channel\_element(); |  |  |
| break; |  |  |
| case ID\_CPE: channel\_pair\_element(); |  |  |
| break; |  |  |
| case ID\_CCE: coupling\_channel\_element(); |  |  |
| break; |  |  |
| case ID\_LFE: lfe\_channel\_element();  break; |  |  |
| break; |  |  |
| case ID\_DSE: data\_stream\_element();  break; |  |  |
| break; |  |  |
| case ID\_PCE: program\_config\_element();  break; |  |  |
| break; |  |  |
| case ID\_FIL: fill\_element();  break; |  |  |
| } |  |  |
| } |  |  |
| byte\_alignment(); |  |  |
| } |  |  |

**id\_syn\_ele** a data element that identifies one of the following syntactic elements:

**Table 4.7 – Syntactic elements**

|  |  |  |  |
| --- | --- | --- | --- |
| ID name | encoding | Abbreviation | Syntactic Element |
| ID\_SCE | 0x0 | SCE | single\_channel\_element() |
| ID\_CPE | 0x1 | CPE | channel\_pair\_element() |
| ID\_CCE | 0x2 | CCE | coupling\_channel\_element() |
| ID\_LFE | 0x3 | LFE | lfe\_channel\_element() |
| ID\_DSE | 0x4 | DSE | data\_stream\_element() |
| ID\_PCE | 0x5 | PCE | program\_config\_element() |
| ID\_FIL | 0x6 | FIL | fill\_element() |
| ID\_END | 0x7 | TERM |  |

**Fill element (FIL)**

The fill element allows to pad data if needed. In addition, it has an extension payload element.

Table 4.8 – Syntax of fill\_element()

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| Fill\_element() |  |  |
| { |  |  |
| cnt = **count;** | **4** | **uimsbf** |
| if (cnt == 15) |  |  |
| cnt += **esc\_count** - 1; | **8** | **uimsbf** |
| while (cnt > 0) { |  |  |
| cnt -= extension\_payload(cnt); |  |  |
| } |  |  |
| } |  |  |

**Audio Data Transport Stream (ADTS)**

An ADTS sequence is a sequence of ADTS frame wherein each frame contains one or more AAC raw blocks.

Table 1.A.9 – Syntax of adts\_sequence()

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| adts\_sequence() |  |  |
| { |  |  |
| while (nextbits() == syncword) { |  |  |
| adts\_frame(); |  |  |
| } |  |  |
| } |  |  |

Table 1.A.10 – Syntax of adts\_frame()

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| adts\_frame() |  |  |
| { |  |  |
| adts\_fixed\_header(); |  |  |
| adts\_variable\_header(); |  |  |
| if (number\_of\_raw\_data\_blocks\_in\_frame == 0) { |  |  |
| adts\_error\_check(); |  |  |
| raw\_data\_block(); |  |  |
| } |  |  |
| else { |  |  |
| adts\_header\_error\_check(); |  |  |
| for( i = 0; i <= number\_of\_raw\_data\_blocks\_in\_frame; i++ ){ |  |  |
| raw\_data\_block(); |  |  |
| adts\_raw\_data\_block\_error\_check(); |  |  |
| } |  |  |
| } |  |  |
| } |  |  |

**Table 1.A.11 – Syntax of adts\_fixed\_header()**

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| adts\_fixed\_header() |  |  |
| { |  |  |
| **syncword;** | **12** | **bslbf** |
| **ID;** | **1** | **bslbf** |
| **layer;** | **2** | **uimsbf** |
| **protection\_absent;** | **1** | **bslbf** |
| **profile\_ObjectType;** | **2** | **uimsbf** |
| **sampling\_frequency\_index;** | **4** | **uimsbf** |
| **private\_bit;** | **1** | **bslbf** |
| **channel\_configuration;** | **3** | **uimsbf** |
| **original\_copy;** | **1** | **bslbf** |
| **home;** | **1** | **bslbf** |
|  |  |  |
| } |  |  |

**Table 1.A.12 – Syntax of adts\_variable\_header()**

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| adts\_variable\_header() |  |  |
| { |  |  |
| **copyright\_identification\_bit;** | **1** | **bslbf** |
| **copyright\_identification\_start;** | **1** | **bslbf** |
| **aac\_frame\_length;** | **13** | **bslbf** |
| **adts\_buffer\_fullness;** | **11** | **bslbf** |
| **number\_of\_raw\_data\_blocks\_in\_frame;** | **2** | **uimsbf** |
| } |  |  |

An ADIF sequence is binary format suitable for storage. It defines an ADIF header prefixing a sequence of AAC raw blocks.

Table 1.A.13 – Syntax of adif\_sequence

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| adif\_sequence() |  |  |
| { |  |  |
| adif\_header(); |  |  |
| byte\_alignment(); |  |  |
| raw\_data\_stream(); |  |  |
| } |  |  |

Table 1.A.14 – Syntax of adif\_header()

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| adif\_header() |  |  |
| { |  |  |
| **adif\_id;** | **32** | **bslbf** |
| **copyright\_id\_present;** | **1** | **bslbf** |
| if (copyright\_id\_present) |  |  |
| **copyright\_id;** | **72** | **bslbf** |
| **original\_copy;** | **1** | **bslbf** |
| **home;** | **1** | **bslbf** |
| **bitstream\_type;** | **1** | **bslbf** |
| **bitrate;** | **23** | **uimsbf** |
| **num\_program\_config\_elements;** | **4** | **bslbf** |
| if (bitstream\_type == ‘0’) { |  |  |
| **adif\_buffer\_fullness;** | **20** | **uimsbf** |
| } |  |  |
| for (i = 0; i < num\_program\_config\_elements + 1; i++) { |  |  |
| program\_config\_element(); |  |  |
| } |  |  |
| } |  |  |

### Feedback from Audio experts at MPEG #150

The full background can be found here: <https://git.mpeg.expert/MPEG/Systems/explorations/-/issues/21>

The main points are:

* It has been discussed that 3GPP SA4 has defined RTP payload format extension mechanism for this purpose, but they would accept new method there is more efficient mechanism available.
* It has been discussed that the profile chosen by 3GPP has the bitstream structure different from the contribution is using
* It has been discussed that it would be difficult apply this kind of approach to AAC decoder implementations due to the structure of bitstream, but MPEG-H is using more flexible bitstream structure in supporting this.
* Approach 2 which is based on the DSE does not work since there is no DSE in AAC-ELD.
* RTP-header extension can be used for the purpose of audio access unit metadata (see e.g. [3GPP TS 26.565](https://www.etsi.org/deliver/etsi_ts/126500_126599/126565/18.01.00_60/ts_126565v180100p.pdf))

### 4.7.2 Possible approaches for metadata carriage in AAC access unit

#### 4.7.2.1 Overview of the approaches

The three approaches are illustrated in the figure below.

|  |  |  |
| --- | --- | --- |
| A screen shot of a computer  AI-generated content may be incorrect.  *Figure 2 - Approach #1* | A black and white screen with text  AI-generated content may be incorrect.  *Figure 3 - Approach #2* | *Figure 4 - Approach #4* |

In approach #1, a new extension type is defined in the fill element. In approach #2, the metadata is carried in a data stream element. Lastly, in approach #3, the metadata is carried in a metadata block within an ADTS sequence.

#### Approach #1: Adding a new extension type payload in the fill element.

This clause highlights the impact in the MPEG-4 part 3.

**Table 4.127 – Values of the extension\_type field**

|  |  |  |
| --- | --- | --- |
| Symbol | Value of extension\_type | Purpose |
| EXT\_FILL | ‘0000’ | bitstream payload filler |
| EXT\_FILL\_DATA | ‘0001’ | bitstream payload data as filler |
| EXT\_DATA\_ELEMENT | ’0010‘ | data element |
| EXT\_DATA\_LENGTH | ‘0011’ | container with explicit length for extension\_payload() |
| EXT\_UNI\_DRC | ’0100‘ | Unified dynamic range control |
| EXT\_AU\_METADATA | ‘0101’ | Access unit metadata |
| EXT\_LDSAC\_DATA | ‘1001’ | LD MPEG Surround |
| EXT\_SAOC\_DATA | ‘1010’ | SAOC |
| EXT\_DYNAMIC\_RANGE | ‘1011’ | dynamic range control |
| EXT\_SAC\_DATA | ‘1100’ | MPEG Surround |
| EXT\_SBR\_DATA | ‘1101’ | SBR enhancement |
| EXT\_SBR\_DATA\_CRC | ‘1110’ | SBR enhancement with CRC |
| EXT\_SAOC\_DE\_DATA | ‘1111’ | SAOC-DE |
| - | all other values | Reserved: These values can be used for a further extension of the syntax in a compatible way. |
| Note: Extension payloads of the type EXT\_FILL or EXT\_FILL\_DATA have to be added to the bitstream payload if the total bits for all audio data together with all additional data are lower than the minimum allowed number of bits in this frame necessary to reach the target bitrate. Those extension payloads are avoided under normal conditions and free bits are used to fill up the bit reservoir. Those extension payloads are written only if the bit reservoir is full. | | |

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| extension\_payload(cnt) |  |  |
| { |  |  |
| **extension\_type**; | **4** | **uimsbf** |
| align = 4; |  |  |
| switch( extension\_type ) { |  |  |
| case EXT\_DYNAMIC\_RANGE: |  |  |
| return dynamic\_range\_info(); |  |  |
| … |  |  |
| case EXT\_AU\_METADATA: |  |  |
| return au\_metadata(); |  |  |
| case EXT\_FIL: |  |  |
| default: |  |  |
| for (i=0; i<8\*(cnt-1)+align; i++) { |  |  |
| **other\_bits[i]**; | **1** | **uimsbf** |
| } |  |  |
| return cnt; |  |  |
| } |  |  |
| } |  |  |
| Note 1: id\_aac is the id\_syn\_ele of the corresponding AAC element (ID\_SCE or ID\_CPE) or ID\_SCE in case of CCE.  Note 2: The extension\_payload() included here shall not have extension\_type == EXT\_DATA\_LENGTH. | | |

|  |  |  |
| --- | --- | --- |
| **Syntax** | **No. of Bits** | **Mnemonic** |
| au\_metadata() { |  |  |
| **TBD** | **tbd** | **tbd** |
| **…** |  |  |
| return n; |  |  |
| } |  |  |

#### Approach #2: Creating a new data structure to be stored in a data stream element.

This clause highlights the impact in the MPEG-4 part 3.

|  |  |  |
| --- | --- | --- |
| **Syntax** | **No. of Bits** | **Mnemonic** |
| access\_unit\_metadata() { |  |  |
| **access\_unit\_metadata\_sync =** 0xBE | **8** | **bslbf** |
| **TBD** | **tbd** | **tbd** |
| **…** |  |  |
| byte\_alignment(); |  |  |
| } |  |  |

data\_stream\_element() Abbreviation DSE. Syntactic element that contains data. Again, there are 16 element\_instance\_tags. There is, however, no restriction on the number of data\_stream\_element()’s with any one instance tag, as a single data stream may continue across multiple data\_stream\_element()’s with the same instance tag.

The DSE may contain a MPEG4\_ancillary\_data element as described in Table 4.12 or an access\_unit\_metadata element as defined in xyz.

#### (abandoned) Approach #3: Defining a metadata data block

This clause highlights the impact in the MPEG-4 part 3.

**Table 1.A.9 – Syntax of adts\_sequence()**

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| adts\_sequence() |  |  |
| { |  |  |
| while (nextbits() == syncword) { |  |  |
| adts\_frame(); |  |  |
| } |  |  |
| } |  |  |

**Table 1.A.10 – Syntax of adts\_frame()**

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| adts\_frame() |  |  |
| { |  |  |
| adts\_fixed\_header(); |  |  |
| adts\_variable\_header(); |  |  |
| if (number\_of\_raw\_data\_blocks\_in\_frame == 0) { |  |  |
| adts\_error\_check(); |  |  |
| If(ID == 1 && profile\_ObjectType == 3) |  |  |
| au\_metadata\_block(); // cf MPEG Systems |  |  |
| else |  |  |
| raw\_data\_block(); |  |  |
| } |  |  |
| else { |  |  |
| adts\_header\_error\_check(); |  |  |
| for( i = 0; i <= number\_of\_raw\_data\_blocks\_in\_frame; i++ ){ |  |  |
| If(ID == 1 && profile\_ObjectType == 3) |  |  |
| au\_metadata\_block(); |  |  |
| else |  |  |
| raw\_data\_block(); |  |  |
| adts\_raw\_data\_block\_error\_check(); |  |  |
| } |  |  |
| } |  |  |
| } |  |  |

**Table 1.A.11 – MPEG-2 Audio profiles and MPEG-4 Audio object types**

|  |  |  |
| --- | --- | --- |
| profile\_ObjectType | MPEG-2 profile (ID == 1) | MPEG-4 object type (ID == 0) |
| 0 | Main profile | AAC Main |
| 1 | Low Complexity profile (LC) | AAC LC |
| 2 | Scalable Sampling Rate profile (SSR) | AAC SSR |
| 3 | ~~(reserved)~~ Access unit metadata block | AAC LTP |

When ID =1 and profile\_ObjectType, the ADTS frame contains au\_metadata\_block and not raw\_data\_block().

#### Approach #4: Generic envelope carried in a fill element with type EXT\_DATA\_ELEMENT and version ANC\_DATA.

This approach does not impact the syntax of MPEG-4 part 3. It consists in defining a generic data envelop with a syncword at the start and placing this envelop inside the extension type EXT\_DATA\_ELEMENT for a ANC\_DATA version.

|  |  |  |
| --- | --- | --- |
| Syntax | No. of bits | Mnemonic |
| extension\_payload(cnt) |  |  |
| { |  |  |
| **extension\_type**; | **4** | **uimsbf** |
| align = 4; |  |  |
| switch( extension\_type ) { |  |  |
| .. |  |  |
| case EXT\_DATA\_ELEMENT: |  |  |
| **data\_element\_version**; | **4** | **uimsbf** |
| switch( data\_element\_version ) { |  |  |
| case ANC\_DATA: |  |  |
| loopCounter = 0; |  |  |
| dataElementLength = 0; |  |  |
| do { |  |  |
| **dataElementLengthPart**; | **8** | **uimsbf** |
| dataElementLength += dataElementLengthPart; |  |  |
| loopCounter++; |  |  |
| } while (dataElementLengthPart == 255); |  |  |
| for (i=0; i<dataElementLength; i++) { |  |  |
| **data\_element\_byte[i]**; | **8** | **uimsbf** |
| } |  |  |
| return (dataElementLength+loopCounter+1); |  |  |
| default: |  |  |
| align = 0; |  |  |
| } |  |  |
| case EXT\_FIL: |  |  |
| default: |  |  |
| for (i=0; i<8\*(cnt-1)+align; i++) { |  |  |
| **other\_bits[i]**; | **1** | **uimsbf** |
| } |  |  |
| return cnt; |  |  |
| } |  |  |
| } |  |  |
| Note 1: id\_aac is the id\_syn\_ele of the corresponding AAC element (ID\_SCE or ID\_CPE) or ID\_SCE in case of CCE.  Note 2: The extension\_payload() included here shall not have extension\_type == EXT\_DATA\_LENGTH. | | |

The exact data envelope could take may form but something along those lines would be expected.

(copied from proposal <https://git.mpeg.expert/MPEG/Systems/explorations/-/issues/21#note_120491> )

metadat\_envelope =

Sync Word - 1 Byte = 0xAB (or whatever syncword works well)

Length field - X Byte

Data - #Length Byte

Parity field 2 Byte

In this scenario, this is still to be discussed where such metadata\_envelope would be defined, in MPEG-4 part 3 or in a specification (WG6) in a WG3 specification.

#### Summary approaches

|  |  |  |  |
| --- | --- | --- | --- |
|  | Approach #1 | Approach #2 | Approach #4 |
| **Syntax location** | In a fill element of a raw block | In a data stream element of a raw block | In a fill element of a raw block as a payload in extension EXT\_DATA\_ELEMENT and version ANC\_DATA |
| **Impacted syntax** | AAC syntax | AAC syntax | No impact on AAC synatax |
| **Size limitation** | If ADTS is used, max value of aac\_frame\_length, i.e., 8191 bytes | If ADTS is used, max value of aac\_frame\_length, i.e., 8191 bytes | Max value of aac\_frame\_length, i.e., 8191 bytes |
| **Decoder can discard** | Yes, assuming unknown extension are discarded | Yes, assuming unknown sync word in data stream element are discarded | Yes, assuming unknown payload after ANC\_DATA are discarded in a fill element. |
| **Impact on bitrate info** | Yes | Yes | Yes |

# 5 Evaluation of different solutions

## 5.1 Issues to study

Based on the background in the earlier clauses of this document, among others the following issues are considered to be study:

* The details of the needed metadata,
  + is the metadata defined in clause 2.2 and 3.5 sufficient to address the use cases under consideration
  + is there any more clarification needed on the details of the metadata
  + Does the information apply in the same manner to audio and video?
* The definition of the metadata should be agnostic to its transport. Considered Options
  + Codec-independent code points (CICP) in 23091
  + MPEG-I Part 7 for Immersive Metadata
* The carriage of the metadata
  + Is the metadata preferably delivered on system level or attached to the media sample?
  + Is SEI message a suitable way to deliver such inband metadata? If so is the proposal in clause 4 sufficient. What additional details need to be defined?
  + How could equivalent information be carried in audio streams?

## 5.2 Metadata Semantics

No additional information is available yet.

## 5.3 Metadata Definition

No additional information is available yet.

## 5.4 Metadata Carriage

As shown in the previous section, the metadata that is carried is highly application specific but needs to be coupled with the media. A rendered pose is associated with a particular video picture or audio frame.

The approach of carrying this metadata in RTP is actually a layer violation. The data is not associated with a particular transport packet but with a whole media access unit.

Another consideration is that it is very likely that in the future, additional application metadata that is associated with the access units will be defined. This will make a solution based on RTP header extensions less feasible due to the impact it will have on the efficiency of the transport compared to carrying it in the payload. For example, it will be very hard for the application to create RTP packets with a highly variable and large amount of RTP header extensions.

# 6 Potential Work in MPEG

## 6.1 General

MPEG may contribute as follows:

* Defining a generic vehicle for carrying application-specific metadata that is frame synchronized and/or associated with media access units will provide a better solution.
* solution should be able to work with all types of media, essentially audio and video, and should enable retrieval of this metadata by the application through appropriate APIs.
* Definition of a unified carriage mechanism for application-specific metadata. The approach should not focus specifically on the split rendering metadata but should rather consider a generic envelope that allows the application to transport application-defined metadata in an opaque manner, while maintaining the accurate synchronization through a solid association with media access units.

As the next steps MPEG solicits input for MPEG#145 on

* additional use cases on the necessity of media-type independent sample metadata
* draft requirements on metadata and the carriage of the metadata
* architectures and call flows supporting the use cases to understand the involved issues when delivering the data over different networks

At MEPG #148, the AHG on exploration on media type independent metadata has been established.

As the next steps MPEG solicits input for MPEG#149 on:

* additional use cases on the necessity of media-type independent sample metadata
* draft requirements on metadata and the carriage of the metadata
* architectures and call flows supporting the use cases to understand the involved issues when delivering the data over different networks
* existing technologies for the carriage of access unit-level metadata in various media bitstreams, including video and audio coding standards

At MPEG #49, two contributions were submitted to the AhG, one commenting on the tentative Working Draft and the second reporting on AAC metadata carriage. In addition, a joint meeting between Audio and Systems was held to present the need for access unit-level metadata and the lack of support of it in audio elementary streams at this point. The contribution [m71642](https://dms.mpeg.expert/doc_end_user/documents/149_Geneva/wg11/m71642-v2-m71642.zip) contains the slides used for this meeting.

As the next steps MPEG solicits input for MPEG#150 on:

* additional use cases on the necessity of media-type independent sample metadata
* draft requirements on metadata and the carriage of the metadata
* architectures and call flows supporting the use cases to understand the involved issues when delivering the data over different networks
* existing technologies for the carriage of access unit-level metadata in various media bitstreams, including video and audio coding standards
* possible high-level design for carrying access unit-level metadata in MPEG-4 AAC as follow-up of the joint Audio and Systems meeting.

## Possible next steps

In order to formalize this potential work in MPEG, a tentative WD has been issued at MPEG #148 and is attached in this exploration document.

# 7 References

[1] OpenXR

[2] Recommendation ITU-R BT.2154-0, High-level system architecture for immersive video for presentation on various types of display devices (2022).

[3] NHK STRL open house 2023.  
<https://www.nhk.or.jp/strl/english/open2023/tenji/5/index.html>

[4] 3GPP TS 26.565