ISO/IEC JTC 1/SC 29/WG 03 N1006

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

**Document type:** Output Document

**Title:** Exploration on Metadata Definition and Carriage for Split Rendering

**Status:** Approved

**Date of document:** 2023-07-21

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

**No. of pages:** 9 (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 N1006**

**July 2023, Geneva, Switzerland**

|  |  |
| --- | --- |
| **Title** | **Exploration on Metadata Definition and Carriage for Split Rendering** |
| **Source** | **WG 03, MPEG Systems** |
| **Status** | **Approved** |
| **Serial Number** | **23033** |

# 1 Introduction

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.

This exploration discusses the background of split-rendering, analysis some first ideas around definition and carriage and metadata and discusses some open issues for further study.

# 2 Background: Split Rendering and OpenXR

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 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 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:

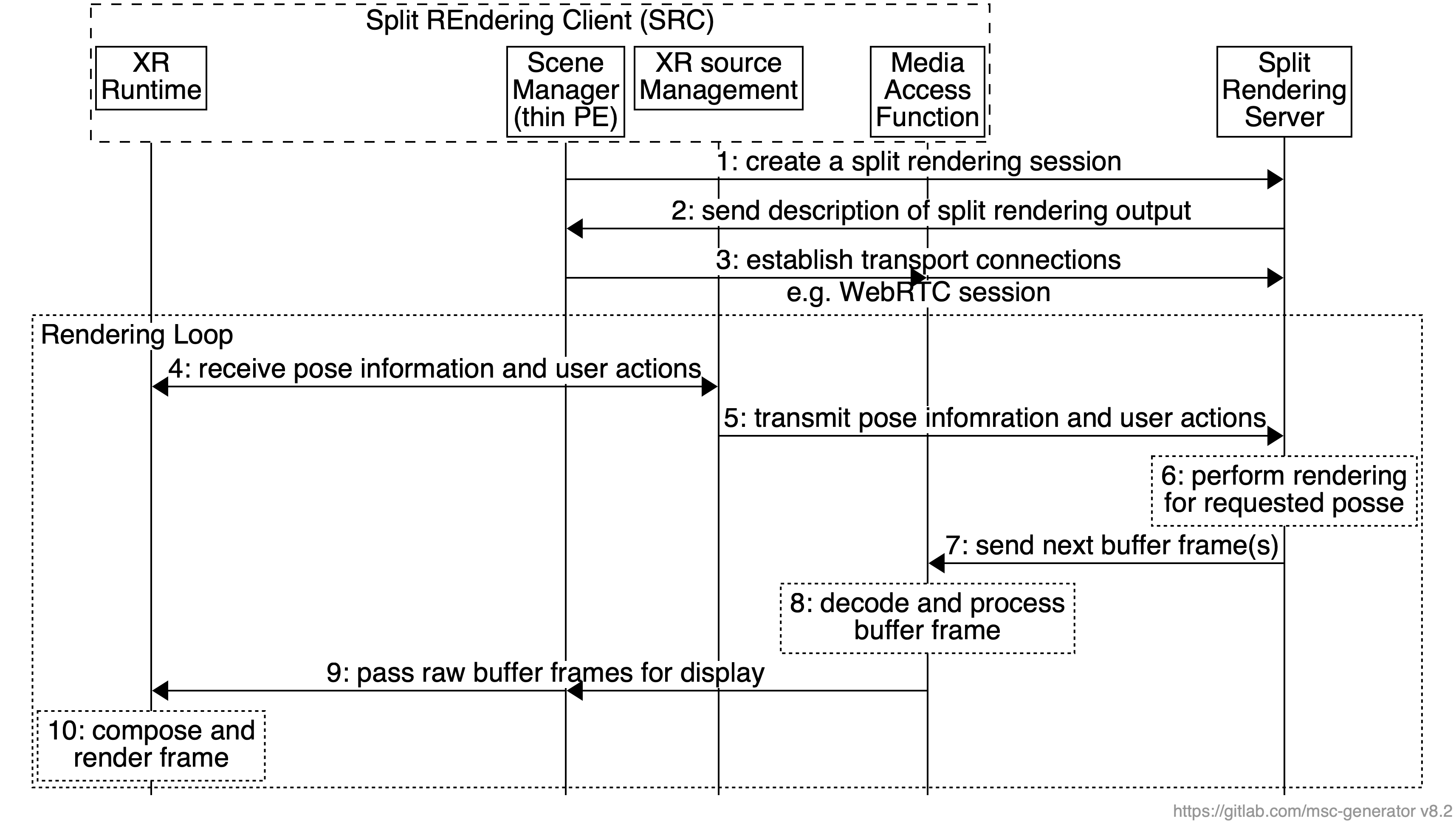


Figure Call flow shows the operation of split rendering as defined in 3GPP TS26.565

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 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 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.

# 4 Carriage of Metadata

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.

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 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) |
| } |  |
| } |  |
| } |  |

# 5 Issues to study further

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 3 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 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?

Input on the above questions as well as additional information is welcome for MPEG#144.