

Enhanced RTMP (V2)

Table of Contents

- [Table of Contents](#)
- [Document Status](#)
- [Documentation Versioning](#)
- [Version Stage Definitions](#)
- [Release Version Disclaimer for Enhanced RTMP](#)
- [Usage License](#)
- [Terminology](#)
- [Abstract](#)
- [Introduction](#)
- [Conventions](#)
- [Simple Data Types](#)
- [RTMP Message Format](#)
- [FLV File Format Overview](#)
- [Enhancements to RTMP and FLV](#)
- [Enhancing onMetaData](#)
- [Reconnect Request](#)
- [Enhanced Audio](#)
- [Enhanced Video](#)
- [Metadata Frame](#)
- [Multitrack Streaming via Enhanced RTMP](#)
- [Enhancing NetConnection connect Command](#)
- [Action Message Format \(AMF\): AMF0 and AMF3](#)
- [Protocol Versioning](#)
- [References](#)

Document Status

Author: Slavik Lozben

Affiliation: [Veovera Software Organization \(VSO\)](#)

Contributors: Adobe, Google, Twitch, Jean-Baptiste Kempf (FFmpeg, VideoLAN), pkv (OBS), Dennis Sädtler (OBS), Xavier Hallade (Intel Corporation), Luxoft, SplitmediaLabs Limited (XSplit), Meta, Michael Thornburgh, Veovera Software Organization

Document Version: **v2-2025-11-21-r1**

General Disclaimer: The features, enhancements, and specifications described in this document are intended for informational purposes only and may not reflect the final implementation. Veovera Software Organization (VSO) does not guarantee the accuracy, completeness, or suitability of this information for any specific purpose. Users are solely responsible for any decisions or implementations based on this document.

VSO reserves the right to refine, update, or enhance any part of this document at its sole discretion, based on technological feasibility, market conditions, or community feedback. VSO shall not be liable for any damages, direct or indirect, resulting from the use of this document.

This document represents a **Release Version** of the enhanced RTMP (E-RTMP) specifications, which is comprehensively specified and stable enough for broad implementation. For a detailed explanation of the Release Version, including its purpose, features, and intended use, please refer to the [Release Version Disclaimer for Enhanced RTMP](#) section later in this document.

Documentation Versioning

Overview

This section outlines the versioning strategy for this specification. This standardized versioning system ensures consistency, traceability, and clarity for implementers by making it easy to determine whether a document introduces new behaviors or includes only editorial updates.

Version identifiers communicate major milestones, publication dates, non-editorial revisions, and the document's development phase, all in a human-readable, chronologically sortable format.

File Naming Convention

Documentation files are named with a clear identifier and a major version number.

Example:

enhanced-rtmp-v2.pdf

Version Information Inside the Document

Each document includes a version identifier using the following format:

Example:

Document Version: v2-2025-04-15-r1

This version string reflects the major version milestone, publication date, non-editorial revision number, and development phase.

Version Format Description

The format for versioning documents is structured as follows:

v<major>-<yyyy-mm-dd>-<phase><revision>

Table: Version format description

Component	Description
v<major>	Major version milestone (e.g. v1, v2). Major versions represent spec maturity but do not necessarily imply breaking changes.
yyyy-mm-dd	The publication date of the document.
<phase>	Document maturity level: a = Alpha, b = Beta, r = Release.
<revision>	Non-editorial revision number. Increments only when changes affect behavior or implementation. Editorial changes (e.g., typo fixes, rewording, format adjustment) do not increment this number. The <revision> counter resets to 1 at each phase transition (alpha → beta → release).

This format supports a clear, linear history. If you're comparing two versions and the revision number (r#) has increased, you can assume there are non-editorial changes that may affect implementation. Editorial-only changes retain the same r# but update the date.

Commit History in GitHub

The document and its [commit history](#) are maintained in the [Enhanced RTMP GitHub repository](#). Although the document version string resets r# at each phase transition, all underlying commits and commit messages are preserved in GitHub, ensuring a complete and auditable record of all changes made.

Version Stage Definitions

We define distinct stages for the development of the E-RTMP protocol specification to indicate its maturity and readiness for implementation. Each stage serves a different purpose for those implementing the protocol:

Alpha Version

- **Purpose:** The alpha stage represents an early, stable version of the protocol specification. It is intended for real-world implementation and feedback collection from developers and implementers who are beginning to build solutions based on the protocol.
- **Features:** While the specification is mostly defined, some aspects may still evolve based on implementation feedback. Breaking changes are possible, but efforts are made to minimize them to ensure stability for early adopters.
- **Audience:** Developers and implementers looking to integrate the protocol in real-world applications who are prepared to adapt to refinements or changes.
- **Stability:** Moderate. The alpha version is considered stable enough for serious implementation, but it is still subject to potential changes that could affect backward compatibility.
- **Documentation Status:** Indicated by the version identifier "a" (e.g., v2-2024-02-26-a1).

Beta Version

- **Purpose:** The beta stage indicates that the protocol specification is nearing its final form, with all core features defined and ready for implementation across diverse environments. This stage focuses on verifying that the protocol works reliably at scale, with extensive real-world testing.
- **Features:** The protocol is comprehensively specified, and any changes at this stage should ideally be non-breaking. These changes may involve optimizations or clarifications to ensure smooth, large-scale deployments, but no core elements of the protocol are expected to be altered.
- **Audience:** Developers and organizations preparing for production use, who are looking to validate their implementations against a near-final version of the protocol.
- **Stability:** High. The beta version is stable enough for broad implementation with the expectation that no significant breaking changes will be introduced.
- **Documentation Status:** Indicated by the version identifier "b" (e.g., v2-2024-02-26-b1).

Release Version (General Availability)

- **Purpose:** The release (General Availability) stage is the finalized version of the protocol specification, fully stable and ready for widespread production use.
- **Features:** The specification is locked, and no breaking changes should occur. Any updates will focus on backward-compatible improvements or bug fixes.
- **Audience:** Developers, implementers, and end-users who need a reliable, long-term stable version for production deployments.
- **Stability:** Highest. The release version ensures stability for production environments with long-term support.
- **Documentation Status:** Indicated by the version identifier "r" (e.g., v2-2024-02-26-r1).

Release Version Disclaimer for Enhanced RTMP

This document represents the **Release Version** of the enhanced Real-Time Messaging Protocol (E-RTMP) specification. All core features are finalized and the specification is considered fully stable for broad implementation across diverse environments.

The protocol is not expected to undergo breaking changes. Future updates to this specification are intended to be backward-compatible, focusing on clarifications, documentation improvements, or optional extensions that do not disrupt existing implementations.

[Veovera Software Organization \(VSO\)](#) provides this document "as is," without warranties, express or implied, including but not limited to suitability for a particular purpose. Users should independently validate their implementations to meet the needs of their specific applications. VSO disclaims all liability for any direct, indirect, incidental, or consequential damages resulting from use of this document.

We welcome community feedback through GitHub issues or other established channels to guide the evolution of the protocol.

This specification is published under the [Apache License, Version 2.0](#), which governs its use, distribution, and modification. Please refer to the [Usage License](#) section later in this document for full licensing terms.

The version identifier (e.g., v2-2024-02-26-r1) reflects the release stage status.

Usage License

Copyright 2022-2024 Veovera Software Organization

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<https://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [BCP 14](#) [[RFC2119](#)] [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here. Definitions below are reproduced from [[RFC2119](#)].

- **MUST:** This word, or the terms "REQUIRED" or "SHALL", means that the definition is an absolute requirement of the specification.
- **MUST NOT:** This phrase, or the phrase "SHALL NOT", means that the definition is an absolute prohibition of the specification.
- **SHOULD:** This word, or the adjective "RECOMMENDED", means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
- **SHOULD NOT:** This phrase, or the phrase "NOT RECOMMENDED", means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
- **MAY:** This word, or the adjective "OPTIONAL", means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

Additionally we add the keyword [[DEPRECATED](#)] to the set of keywords above.

- **DEPRECATED:** This word means a discouragement of use of some terminology, feature, design, or practice, typically because it has been superseded or is no longer considered efficient or safe, without completely removing it or prohibiting its use. Typically, deprecated materials are not completely removed to ensure legacy compatibility or back-up practice in case new methods are not functional in an odd scenario. It can also imply that a feature, design, or practice will be removed or discontinued entirely in the future.

Abstract

In the rapidly evolving media streaming landscape, there is a pressing need to update legacy protocols to align with modern technological standards. The Real-Time Messaging Protocol [[RTMP](#)] and Flash Video [[FLV](#)] file format, introduced in 2002, have been pivotal and continue to be vital especially in live broadcasting. Despite RTMP widespread use, it has shown signs of aging, particularly in the lack of support for contemporary video codecs (e.g., VP8, VP9, HEVC, AV1) and audio codecs (Opus, FLAC, AC-3, E-AC-3). Recognizing this, Veovera Software Organization (VSO), in collaboration with industry giants like Adobe, YouTube, and Twitch, and other key stakeholders, has embarked on a mission to rejuvenate RTMP, ensuring it meets the demands of contemporary streaming needs.

This document details the comprehensive enhancements made to the RTMP and FLV specifications, aimed at revitalizing the technology for current and future media demands. Our strategic approach prioritizes innovation while maintaining backward compatibility, thereby augmenting RTMP's utility without undermining existing infrastructures. Some of the key advancements include:

- **Advanced Audio Codecs:** Integration of codecs like AC-3, E-AC-3, Opus, and FLAC to meet diverse audio quality and compression needs, ensuring compatibility with modern systems.
- **Multichannel Audio Configurations:** Support for multichannel audio to enhance auditory experiences without compromising existing setups.
- **Advanced Video Codecs:** Introduction of codecs such as VP8, VP9, HEVC and AV1 with HDR support to meet modern display and content standards.
- **Video Metadata:** Expansion of `VideoPacketType.Metadata` to support a broader range of video metadata types.
- **FourCC Signaling:** Inclusion of FourCC signaling for advanced codecs mentioned above, as well as for legacy codecs such as AVC, AAC, and MP3.
- **Multitrack Capabilities:** New audio and video multitrack capabilities for concurrent management and processing of multiple media streams, enhancing media experiences.
- **Reconnect Request Feature:** A new Reconnect Request feature improves connection stability and resilience.
- **Timestamp Precision:** Introduction of nanosecond precision offsets, ensuring enhanced synchronization and compatibility across diverse media formats such as MP4, M2TS, and Safari's Media Source Extensions, without altering the core RTMP timestamps.
- **ModEx Signal:** A signaling mechanism that lets packets carry modifiers or extensions, enabling precise control over timing, metadata, and other media stream behaviors.

The additional audio and video codecs supported by enhanced RTMP are summarized in the following table:

Table: Additional audio and video codecs for E-RTMP

Additional Audio Codec	Notes
AC-3	AC-3 and E-AC-3 have significantly influenced the surround sound market by offering versatile and scalable audio solutions for both physical and streaming media. Their balance of complexity and performance makes them enduring standards in multichannel audio technology.
E-AC-3	
Opus	
FLAC	Popular in both hardware and software streaming solutions, the [WebCodecs] audio codec registry also includes support for these widely used audio formats.
AAC (added FOURCC signaling)	
MP3 (added FOURCC signaling)	
Additional Video Codec	
AVC (a.k.a., H.264, added FOURCC signaling)	
HEVC (a.k.a., H.265)	
VP8 (webRTC officially supports this codec)	Popular in both hardware and software streaming solutions, the [WebCodecs] video codec registry also includes support for these widely used video formats.
VP9	

These strategic enhancements position RTMP as a robust, future-proof standard in the streaming technology arena. Veovera is committed to open collaboration and values community input, believing that protocols and standards should be open and free to foster innovation and create a thriving ecosystem. Companies can capitalize on solutions built around open standards; the more popular and accessible a protocol is, the stronger the foundation for developing compelling solutions. A standard's popularity fuels adoption, allowing companies to leverage its widespread use. In contrast, fragmentation caused by proprietary protocols hampers industry growth, while open standards empower everyone to innovate freely, creating a healthier marketplace. E-RTMP's shift toward openness aligns with the principles of open standards, emphasizing its potential to become a foundational technology. We encourage participation in the ongoing development process through our [GitHub repository](#), where you can access detailed documentation, contribute to the project, and share insights to foster a vibrant ecosystem around enhanced E-RTMP.

Introduction

This document describes enhancements to legacy [\[RTMP\]](#) and legacy [\[FLV\]](#), introducing support for new media codecs, HDR capability, and more. A primary objective is to ensure these enhancements do not introduce breaking changes for established clients or the content they stream. As such, legacy RTMP and legacy FLV specifications remain integral to the RTMP ecosystem. While this updated specification aims to minimize redundancy with previous versions, when combined with previous-generation documentation, it provides a comprehensive overview of the RTMP solution. We've drawn from several legacy references, which are as follows:

- Adobe legacy [\[RTMP\]](#) specification
- Adobe legacy [\[FLV\]](#) specification
- Additional [\[LEGACY\]](#) specifications

Conventions

This document employs certain conventions to convey particular meanings and requirements. The following section outlines the notation, terminology, and symbols used throughout to ensure clarity and consistency. These conventions provide insight into the ethos of how the E-RTMP specification has been crafted and should be interpreted.

- **E-RTMP:** refers to a series of improvements made to the legacy Real-Time Messaging Protocol [\[RTMP\]](#), originally developed by Adobe. While "enhanced RTMP" may be used descriptively, the preferred and consistent name for the protocol is **E-RTMP**, which serves as a consistent identifier that distinguishes the updated protocol from the legacy RTMP specification. Endorsed by Adobe and widely adopted across the industry, E-RTMP serves as the current standard for RTMP-based solutions and includes enhancements to both RTMP and the legacy [\[FLV\]](#) formats. The name E-RTMP refers to an evolving protocol and does not correspond to any single release or version. To avoid confusion, alternate forms such as *ERTMP*, *eRTMP*, *Enhanced-RTMP*, or *EnhancedRTMP* should be avoided. The lowercase hyphenated form *e-rtmp* is acceptable in URLs, folder names, and other technical contexts where lowercase formatting is conventional.

- **Pseudocode:** Pseudocode has been provided to convey logic on how to interpret the E-RTMP binary format. The code style imitates a cross between TypeScript and C. The pseudocode was written in TypeScript and validated using VSCode to ensure correct syntax and catch any minor typographical errors. Below are some further explanations:

- **Enumerations define valid values for fields that appear in the encoded bitstream.** In contrast, script-level may use textual names or key/value maps, consistent with historical SCRIPTDATA and ActionScript object model.
- Pseudo variables are named in a self-descriptive manner. For instance:

```
videoCommand = UI8 as VideoCommand`
```

The line above indicates that an unsigned 8-bit value is read from the bitstream. The legal values correspond to the enumerations within the **VideoCommand** set, and the pseudo variable **videoCommand** now holds that value.

- The pseudocode is written from the point of view of reading (a.k.a., parsing) the bitstream. If you are writing the bitstream, you can swap source with destination variables.
- E-RTMP typically employs camelCase naming conventions for variables. In contrast, the naming convention for legacy RTMP specification is usually preserved as is.
- Handshake and [Enhancing NetConnection connect command](#): The E-RTMP specification generally prioritizes the client's perspective over that of the server. To shift this focus and view the interaction from the server's side, the server should echo back certain enhancement information.

When the client informs the server of the enhancements it supports via the **connect** command, the server processes this command and responds using the same transaction ID. The server's response string will be one of the following: **_result**, **_error**, or a specific method name. A command string of **_result** or **_error** indicates a response rather than a new command.

During this response, the server will include an object containing specific properties as one of the arguments to **_result**. It is at this point that the server should indicate its support for E-RTMP features. Specifically, the server should denote its capabilities through attributes such as **videoFourCcInfoMap**, **capsEx**, and other defined properties.

- The ethos of this pseudocode is to provide a high-level overview of the data structures and operations taking place on the wire. While it accurately represents the bytes being transmitted, it's important to note that the logic is not exhaustive. Specifically, this pseudocode does not cover all possible cases, nor does it always include items such as initialization logic, looping logic or error-handling mechanisms. It serves as a foundational guide that can be implemented in various ways, depending on specific needs and constraints.

- **Unrecognized value:** If a value in the bitstream is not understood, the logic must fail gracefully in a manner appropriate for the implementation.
- **Table naming:** Each table in the document is named according to the specific content or subject it is describing.
- **Bitstream optimization:** One of the guiding principles of E-RTMP is to optimize the number of bytes transmitted over the wire. While minimizing payload overhead is a priority, it is sometimes more important to simplify the logic or enhance extensibility. For example, although more optimal methods for creating a codec ID than using FOURCC may exist, such approaches could render the enhancement non-standard and more challenging to extend and maintain in the future.

- **Capitalization rules:** Another guiding principle in the E-RTMP is the standardization of capitalization for types. The original documentation capitalized types such as Number, String, and Boolean, and even included various other spellings. The E-RTMP adopts lowercase spelling for terms, such as number, string, and boolean. This change emphasizes that these types are simple, not objects.
- **ECMA Array vs Object:** In the world of AMF (Action Message Format), both ECMA Array and Object are used to store collections of properties. A property is simply a pairing of a name with a value. In enhanced RTMP, the term Object is specifically used to indicate the Object Type. In the past, people have sometimes used ECMA Array and Object as if they were the same thing. However, for better coding practices, it's recommended to use Object when you're creating AMF data. When you're reading or decoding AMF data, you should be prepared to handle either ECMA Array or Object for greater flexibility and robustness.
- **Default values:** Unless explicitly called out, there should be no assumptions made regarding default values, such as null or undefined.
- **Legacy vs. Enhanced Properties:** In the documentation, an effort has been made to distinguish between legacy properties and newly defined ones through color coding, such as using bold text or different background colors for enhancements. While this color coding is not guaranteed to be consistent, the distinctions between values defined in E-RTMP should be readily apparent.
- **Capability flags:** The capabilities flags, exchanged during a connect handshake, may not cover all possible functionalities. For instance, a client might indicate support for multitrack processing without specifying its ability to encode or decode multitrack streams. In scenarios where a client, capable of issuing a play command, declares multitrack support, it MUST be equipped to handle the playback of such streams. Similarly, if a client is aware of the server's multitrack capabilities, it MAY opt to publish a multitrack stream.
- **Quotation Marks and Emphasis Guidelines:** Ultimately, the context should drive the meaning, but we make an effort to leverage quotation marks and emphasis (i.e., **bold**) to maintain readability. We aim to avoid syntactic sugar as much as possible to ensure the document remains straightforward, easy to read, scan, and understand. The conventions for using double quotes ("), back quotes (`), and emphasis in this document to ensure clarity and consistency are as follows:
 - **Double quotes are used for:** direct quotations, titles of short works, and when referencing a specific term or phrase.
 - **Back quotes are used for:** code snippets, commands, or technical terms.
 - **Bold is used for:** emphasis on important terms or phrases. Sometimes, back quotes and bold can be interchanged for ease of reading.

Simple Data Types

The following data types are used in [\[RTMP\]](#) bitstreams and [\[FLV\]](#) files. FOURCC was introduced to support E-RTMP.

Table: Simple data types

Type	Definition
0x...	Hexadecimal value
UB[n]	Bit field with unsigned n-bit integer, where n is in the range 1 to 31, excluding 8, 16, 24
FOURCC	Four-character ASCII code, such as "av01", encoded as UI32
SI8	Signed 8-bit integer
SI16	Signed 16-bit integer
SI24	Signed 24-bit integer

SI32	Signed 32-bit integer
UI8	Unsigned 8-bit integer
UI16	Unsigned 16-bit integer
UI24	Unsigned 24-bit integer
UI32	Unsigned 32-bit integer
xxx[]	Array of type xxx. Number of elements to be inferred
xxx[n]	Array of n elements of type xxx
[xxx]	Array of one element of type xxx

^ ^>

Note: Unless specifically called out, multi-byte integers SHALL be stored in big-endian byte order

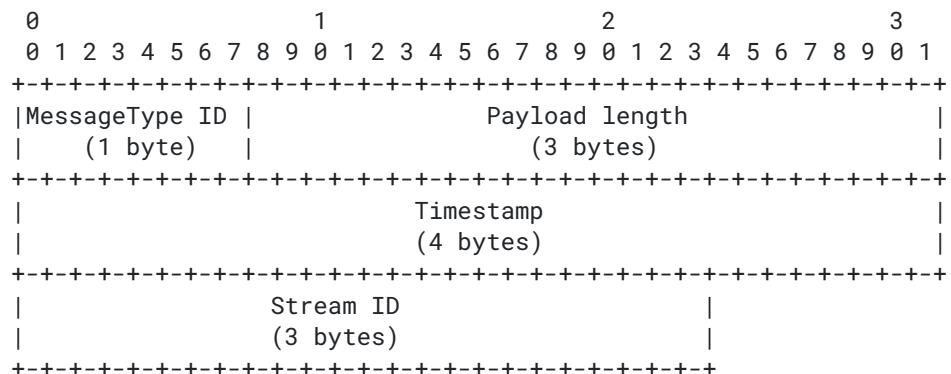
^ ^>

RTMP Message Format

Adobe's Real-Time Messaging Protocol [RTMP] is an application-level protocol designed for the multiplexing and packetizing of multimedia streams—such as audio, video, and interactive content, for transmission over network protocols like TCP. A fundamental feature of RTMP is the Chunk Stream, which facilitates the multiplexing, packetizing, and prioritization of messages, integral to the protocol's real-time capabilities.

The legacy RTMP specification in [Section 6.1](#) elaborates on the RTMP Message Format, providing precise encoding guidelines for the RTMP message header, inclusive of field widths and byte order. However, this portrayal might be somewhat confusing because RTMP messages, when transported over the Chunk Stream, don't literally conform to this depicted format. An RTMP Message is divided into two principal components: a message virtual header and a message payload. The "virtual" descriptor indicates that while RTMP messages are carried within the RTMP Chunk Stream, their headers are conceptually encoded as Chunk Message Headers. When these are decoded from the RTMP Chunk Stream, the underlying transport layer, the resulting format is to be understood as a virtual header. This abstract representation aligns with the structured format and semantics detailed in the legacy RTMP specification. Detailed next is the format of the message virtual header and some additional related information.

- Message virtual header



- There are two message types reserved for media messages:
 - The message type value of 8 is reserved for audio message
 - The message type value of 9 is reserved for video messages
- The message payload follows the header and may contain various types of content, such as compressed audio or video data. RTMP itself does not recognize or process the payload's content. If new codec types are to be added, they must be defined where the actual payload internals are outlined. FLV is a container file format where the specifics of the AV payload, including the codecs, are defined.
- Please refer to the legacy RTMP specification (in various locations) and the legacy [\[FLV\]](#) specification (Annex E) for details on the endianness (a.k.a., byte order) of the data format on the wire.

FLV File Format Overview

An [\[FLV\]](#) file is a container for AV (Audio and Video) data. The file consists of alternating back-pointers and tags, each accompanied by data related to that tag. Each **TagType** within an FLV file is unsigned and defined by 5 bits. **AUDIODATA** has a **TagType** of 8, and **VIDEODATA** has a **TagType** of 9.

^ ^ >

Note: Each **TagType** corresponds directly to the same **MessageType ID**, defined by UI8, in the [\[RTMP\]](#) specification. This alignment is intentional.

^ ^ >

TagType values of 8 or 9 are accompanied by an **AudioTagHeader** or **VideoTagHeader** respectively. While RTMP is commonly associated with FLV, it is important to note that RTMP is a protocol, whereas FLV is a file container format. This distinction is why they were originally defined in separate specifications. This enhancement specification aims to improve both RTMP and FLV.

Pre 2023 AudioTagHeader Format

Below is the **AudioTagHeader** format for the legacy FLV specification:

Table: FLV specification [AudioTagHeader](#)

Field	Type	Comment
SoundFormat	UB[4]	Format of SoundData. The following values are defined: 0 = Linear PCM, platform-endian 1 = ADPCM 2 = MP3 3 = Linear PCM, little-endian 4 = Nellymoser 16 kHz mono 5 = Nellymoser 8 kHz mono 6 = Nellymoser 7 = G.711 A-law logarithmic PCM 8 = G.711 mu-law logarithmic PCM 9 = Reserved 10 = AAC

		11 = Speex 12 = Reserved 13 = Reserved 14 = MP3 8 kHz 15 = Device-specific sound Formats 7, 8, 14, and 15 are reserved. AAC is supported in Flash Player 9,0,115,0 and higher. Speex is supported in Flash Player 10 and higher.
SoundRate	UB[2]	Sampling rate. The following values are defined: 0 = 5.5 kHz 1 = 11 kHz 2 = 22 kHz 3 = 44 kHz
SoundSize	UB[1]	Size of each audio sample. This parameter only pertains to uncompressed formats. Compressed formats always decode to 16 bits internally. 0 = 8-bit samples 1 = 16-bit samples
SoundType	UB[1]	Mono or stereo sound 0 = Mono sound 1 = Stereo sound
AACPacketType	IF SoundFormat == 10 UI8	The following values are defined: 0 = AAC sequence header 1 = AAC raw

Pre 2023 VideoTagHeader Format

Below is the **VideoTagHeader** format for the legacy FLV specification:

Table: FLV specification [VideoTagHeader](#)

Field	Type	Comment
Frame Type	UB[4]	Type of video frame. The following values are defined: 1 = key frame (for AVC, a seekable frame) 2 = inter frame (for AVC, a non-seekable frame) 3 = disposable inter frame (H.263 only) 4 = generated key frame (reserved for server use only) 5 = video info/command frame
CodecID	UB[4]	Codec Identifier. The following values are defined: 2 = Sorenson H.263 3 = Screen video 4 = On2 VP6 5 = On2 VP6 with alpha channel 6 = Screen video version 2 7 = AVC
AVC.PacketType	IF CodecID == 7 UI8	The following values are defined: 0 = AVC sequence header 1 = AVC NALU

		2 = AVC end of sequence (lower level NALU sequence ender is not REQUIRED or supported)
CompositionTime	IF CodecID == 7 SI24	<pre> IF AVCPacketType == 1 Composition time offset ELSE 0 See ISO/IEC 14496-12, 8.15.3 for an explanation of composition times. The offset in an FLV file is always in milliseconds. </pre>

Enhancements to RTMP and FLV

Within the following sections, this document provides a comprehensive overview of the enhancements made to [RTMP] and [FLV]. Together, these improvements constitute the enhanced RTMP also known as E-RTMP. These enhancements are discussed in detail, highlighting their impact and benefits.

Enhancing onMetaData

[FLV] metadata SHALL be encapsulated within a [SCRIPTDATA] segment, which includes a [ScriptTagBody] encoded in the Action Message Format (AMF). Importantly, this metadata SHALL always remain unencrypted, even when the FLV content itself is encrypted. This design choice is essential for allowing various FLV parsers to successfully stream the FLV content and for enabling media players to provide contextual information to the user.

The **ScriptTagBody** is structured to encapsulate method invocations. It consists of an item containing a method name (e.g., **onMetaData**) along with a corresponding set of arguments.

To signal FLV metadata, the item within the **ScriptTagBody** MUST encapsulate the method name **onMetaData**, along with a single argument of type ECMA array. This array holds metadata properties, the availability of which may vary depending on the software used to create the FLV. Typical **onMetaData** argument properties include, but are not limited to:

Table: Typical properties found in the **onMetaData** argument object

Property	Type	Comment
audiocodecid	number	<p>Audio codec ID used in the file: See AudioTagHeader of the legacy [FLV] specification for available CodecID values.</p> <p>When [FourCC] is used to signal the codec, this property is set to a FOURCC value. Note: A FOURCC value is big-endian relative to the underlying ASCII character sequence (e.g., "Opus" == 0x4F70573 == 1332770163.0).</p>
audiodatarate	number	Audio bitrate, in kilobits per second

audiodelay	number	Delay introduced by the audio codec, in seconds
audiosamplerate	number	Frequency at which the audio stream is replayed
audiosamplesize	number	Number of bits used to represent each audio sample
canSeekToEnd	boolean	Indicating the last video frame is a key frame
creationdate	string	Creation date and time
duration	number	Total duration of the file, in seconds
filesize	number	Total size of the file, in bytes
framerate	number	Number of frames per second
height	number	Height of the video, in pixels
stereo	boolean	Indicates stereo audio
videocodecid	number	<p>Video codec ID used in the file: See VideoTagHeader of the legacy [FLV] specification for available CodecID values.</p> <p>When [FourCC] is used to signal the codec, this property is set to a FOURCC value. Note: A FOURCC value is big-endian relative to the underlying ASCII character sequence (e.g., "av01" == 0x61763031 == 1635135537.0).</p>
videodatarate	number	Video bitrate, in kilobits per second
width	number	Width of the video, in pixels

audioTrackIdInfoMap	Object	<p>audioTrackIdInfoMap and videoTrackIdInfoMap are objects used to provide per-track metadata for additional audio and video tracks beyond the default track. Each object uses a trackId as a key to describe the characteristics of a specific non-default track. The default track for each media type uses trackId = 0 and is described by the top-level onMetaData fields for that media type. Tracks with trackId values 1, 2, 3, ... represent additional variants.</p> <ul style="list-style-type: none"> • Key-Value Structure: <ul style="list-style-type: none"> ◦ Keys: Each key represents a trackId for a non-default track. Keys begin at 1, since the default track (trackId 0) is represented by the top-level onMetaData fields. ◦ Values: Each value is an object containing metadata that describes the characteristics of that specific track. • Properties of Each Track Object: These properties describe track-level attributes for each additional track. These attributes often differ from the top-level onMetaData fields, for example, a different bitrate, resolution, codec, channel count, or other media properties. Implementations may choose either approach: <ul style="list-style-type: none"> ◦ Provide only the properties that differ from the top-level defaults (delta-style), or ◦ Repeat shared fields so that each track entry is a complete per-track descriptor. Both approaches are valid and interoperable. Typical fields include, but are not limited to: <ul style="list-style-type: none"> ◦ For videoTrackIdInfoMap: width, height, videodatarate, codec identifier, and related video characteristics. ◦ For audioTrackIdInfoMap: audiodatarate, channels, samplerate, codec identifier, and related audio characteristics. • Purpose: The purpose of these maps is to describe the characteristics of additional tracks so that encoders, servers, and players can distinguish between variants and, when needed, apply track selection or processing logic. These maps supplement the top-level onMetaData fields by providing per-track metadata for multitrack scenarios such as alternative codecs, different qualities, or configuration variants. <p>This structure provides a framework for detailed customization and control over the media tracks, ensuring optimal management and delivery across various types of content and platforms.</p> <p>Example:</p> <pre>// TrackId 0 is the default track and is described by the top-level // onMetaData fields for each media type. Additional tracks begin at TrackId 1. var videoTrackIdInfoMap = { 1: { width: 1024, height: 768, videodatarate: 2000, videocodecid: makeFourCc("av01"), }, 2: { width: 1280, height: 720, videodatarate: 3000, videocodecid: makeFourCc("av02"), }, 3: { width: 1920, height: 1080, videodatarate: 5000, videocodecid: makeFourCc("av03"), } };</pre>
---------------------	--------	--

videoTrackIdInfoMap	<pre> width: 3840, height: 2160, videodatarate: 30000, videocodecid: makeFourCc("avc1"), }, } var audioTrackIdInfoMap = { 1: { audiodatarate: 256, channels: 2, samplerate: 44100, audiocodecid: makeFourCc("mp4a"), }, 2: { audiodatarate: 320, channels: 2, samplerate: 48000, audiocodecid: makeFourCc("Opus"), }, } </pre>
---------------------	---

^ ^ >

Note:

- The properties **audiocodecid** and **videocodecid** have been enhanced to support FOURCC (Four-byte ASCII code) values. These values are interpreted as UI32 (e.g., "av01").
- The properties **audioTrackIdInfoMap** and **videoTrackIdInfoMap** are new.

^ ^ >

Reconnect Request

Objective

[[RTMP](#)] packetizes multimedia streams using a suitable transport protocol, typically a persistent TCP connection. There are instances when a streaming platform may request the streaming client to reconnect, such as:

- When live streaming servers undergo updates.
- When there's a need to redirect the client to a different server instance, ensuring optimal load balancing and precise geolocation mapping.

To accommodate these needs, a **NetConnection.Connect.ReconnectRequest** status event has been introduced as part of the **NetConnection onStatus** command.

NetConnection Commands

NetConnection establishes a bidirectional link between a client and a server, allowing for asynchronous Remote Procedure Calls (RPCs). The following commands (a.k.a., predefined RPCs) can be issued via **NetConnection**:

- **connect**
- **createStream**
- **deleteStream**
- **onStatus**

The **onStatus** command has been enhanced to include the capability to request a client to reconnect. Servers can issue an **onStatus** command to prompt clients to adapt to changes in **NetConnection** status. The structure of this command, as relayed from the server to the client, is outlined below:

Table: Server to client, **NetConnection onStatus** command

Field Name	Type	Description
Command Name	string	Name of the command. Set to onStatus
Transaction ID	number	Transaction ID set to 0. (i.e., no response needed)
Command Object	null	There is no command object for onStatus command.
Info Object	Object	An AMF-encoded object, the properties of which are utilized by the onStatus command. The Info Object provides information about the status of the current connection.

The following is a description of AMF-encoded name-value pairs in the Info Object for the **onStatus** command when handling reconnect. It MAY contain other properties as appropriate to the client.

Table: Info Object parameter for **onStatus** command when handling reconnect

Property	Type	Description	Example Value
tcUrl (optional)	string	Absolute or relative URI reference of the server to which to reconnect. If not specified, use the tcUrl for the current connection. A relative URI reference should be resolved relative to the tcUrl for the current connection.	1. rtmp://foo.mydomain.com:1935/realtimeapp 2. rtmp://127.0.0.1/realtimeapp 3. //192.0.2.0/realtimeapp 4. /realtimeapp
code	string	A string identifying the event that occurred. To reconnect code MUST be set to NetConnection.Connect.ReconnectRequest	NetConnection.Connect.ReconnectRequest
description (optional)	string	A string containing human-readable information about the message. Not every information object includes this property.	The streaming server is undergoing updates.
level	string	A string indicating the severity of the event. To reconnect the level MUST be set to status .	status

Message Flow When Handling `NetConnection.Connect.ReconnectRequest`

1. Prior to the shutdown of the live streaming server or when the server intends to remap the client to another server instance, it dispatches an `onStatus` command to the client with a `code` of `NetConnection.Connect.ReconnectRequest`. If the server aims to remap the client, it MUST set the `tcUrl` property in the Info Object. In order to avoid a disruption, the server managing the original connection (commonly referred to as the "old server") SHOULD continue processing messages from the client until the client disconnects.
2. When the client receives the `NetConnection.Connect.ReconnectRequest` event, it persists in streaming to/from the current server up to the next appropriate media boundary, such as a keyframe. Subsequently, it establishes a connection with a new server and disconnects from the old server. If the Info Object includes the `tcUrl` property, the client uses this URL for the reconnection process. Absent this property, the client defaults to the `tcUrl` for the current connection.
3. While the client can establish a new connection before severing the original one, it SHOULD exercise caution to ensure the Quality of Service (QoS) is not compromised.

The capability to support the `NetConnection.Connect.ReconnectRequest` event becomes evident during the initial connect phase. Detailed guidelines for signaling reconnect ability can be found in the [Enhancing `NetConnection.connect` Command](#) section.

Detailed Overview of the `onStatus` Command for `NetConnection`

The server-to-client `onStatus` command for `NetConnection`, serves a crucial function within the RTMP framework. Though the legacy RTMP specification may not have detailed this command, the goal here is to offer an overview for a better understanding.

Both clients and servers can initiate RPCs at the receiving end, with some RPCs being predefined as commands. `onStatus` stands out as one such essential command.

When using the `onStatus` command, the goal is to inform the client about the status of the connection. Each dispatched command message comprises the following elements:

- **Command Name:** type string
- **Transaction ID:** type number
- **Command Object** (set to null when dispatching an `onStatus` command): type Object
- **Info Object** (which can be viewed as Optional Arguments): type Object

Both the Command Object and the Info Object offer additional context and details for the command. The `onStatus` command is triggered whenever there's a status change or an error concerning the `NetConnection`. To handle this information, you should define a callback function.

```
```js
// Sample pseudocode for the onStatus callback function
nc.onStatus = function(infoObject) {
 // Handle the status change or error here.
}
````
```

infoObject is an AMF-encoded object with properties that provide information about the status of a **NetConnection**. It contains at least the following three properties, but MAY contain other properties as appropriate to the client.

Table: infoObject for onStatus command

| Property | Type | Description | Example Value |
|------------------------|--------|--|-----------------------------------|
| code | string | A string identifying the event that occurred. | NetConnection.Connect.Success |
| description (optional) | string | A string containing human-readable information about the message. Not every information object includes this property. | The connection attempt succeeded. |
| level | string | There are three established values for level: status , warning , and error . | status |

The table below provides examples of **code**, **level**, and **description** property values. Please note that this is not an exhaustive list, and not all entries may apply to every type of client. Additionally, the **description** property values included are merely illustrative examples; developers are responsible for conveying the appropriate meaning in their specific solutions.

Table: code, level and description values for infoObject used by onStatus

| Code | Level | Description |
|---|---------------|---|
| NetConnection.Call.Failed | error | The NetConnection.call() method was not able to invoke the server-side method or command. |
| NetConnection.Connect.AppShutdown | error | The application has been shut down (for example, if the application is out of memory resources and must shut down to prevent the server from crashing) or the server has shut down. |
| NetConnection.Connect.Closed | status | The connection was closed successfully. |
| NetConnection.Connect.Failed | error | The connection attempt failed. |
| NetConnection.Connect.Rejected | error | The client does not have permission to connect to the application. |
| NetConnection.Connect.Success | status | The connection attempt succeeded. |
| NetConnection.Connect.ReconnectRequest | status | The server is requesting the client to reconnect. |
| NetConnection.Proxy.NotResponding | error | The proxy server is not responding. See the ProxyStream class. |

Enhanced Audio

The **AudioTagHeader** has been extended to define additional audio codecs, multichannel audio, multitrack capabilities, signaling support, and additional miscellaneous enhancements, while ensuring backward compatibility. This extension is termed the **ExAudioTagHeader** and is designed to be future-proof, allowing for the definition of additional audio codecs, features, and corresponding signaling.

During the parsing process, the logic MUST handle unexpected or unknown elements gracefully. Specifically, if any critical signaling or flags (e.g., `AudioPacketType` and `AudioFourCc`) are not recognized, the system MUST fail in a controlled and predictable manner.

^ ^ >

Important: A single audio message for a unique timestamp may include a batch of `AudioPacketType` values (e.g., multiple `trackId` values). When parsing an audio message, the bitstream MUST be processed completely to ensure all payload data has been handled.

^ ^ >

Table: Extended AudioTagHeader

| Description Of Bitstream | Enumerated Types |
|--|--|
| <pre> soundFormat = UB[4] as SoundFormat if (soundFormat != SoundFormat.ExHeader) { // See AudioTagHeader of the legacy [FLV] specification for for detailed format // of the four bits used for soundRate/soundSize/soundType // // NOTE: soundRate, soundSize and soundType formats have not changed. // if (soundFormat == SoundFormat.ExHeader) we switch into FOURCC audio mode // as defined below. This means that soundRate, soundSize and soundType // bits are not interpreted, instead the UB[4] bits are interpreted as an // AudioPacketType soundRate = UB[2] soundSize = UB[1] soundType = UB[1] } </pre> | <pre> enum SoundFormat { LPcmPlatformEndian = 0, AdPcm = 1, Mp3 = 2, LPcmLittleEndian = 3, Nellymoser16KMono = 4, Nellymoser8KMono = 5, Nellymoser = 6, G711ALaw = 7, G711MuLaw = 8, ExHeader = 9, // new, used to signal FOURCC mode Aac = 10, Speex = 11, // 12 - reserved // 13 - reserved Mp3_8K = 14, Native = 15, // Device specific sound } </pre> |
| ExAudioTagHeader Section | |
| Note: ExAudioTagHeader is present if (soundFormat == SoundFormat.ExHeader) | |
| Description Of Bitstream | Enumerated Types |
| <pre> /// // process ExAudioTagHeader // processAudioBody = false if (soundFormat == SoundFormat.ExHeader) { processAudioBody = true // Interpret UB[4] bits as AudioPacketType instead of sound rate, size, and type. audioPacketType = UB[4] as AudioPacketType // at byte boundary after this read // Process each ModEx data packet while (audioPacketType == AudioPacketType.ModEx) { // Determine the size of the packet ModEx data (ranging from 1 to 256 bytes) modExDataSize = UI8 + 1 // If maximum 8-bit size is not sufficient, use a 16-bit value if (modExDataSize == 256) { modExDataSize = UI16 + 1; } // Fetch the packet ModEx data based on its determined size modExData = UI8[modExDataSize] // fetch the AudioPacketModExType audioPacketModExType = UB[4] as AudioPacketModExType // Update audioPacketType } } </pre> | <pre> enum AudioPacketType { SequenceStart = 0, CodedFrames = 1, // RTMP includes a previously undocumented "audio silence" message. // This silence message is identified when an audio message contains // a zero-length payload, or more precisely, an empty audio message // without an AudioTagHeader, indicating a period of silence. The // action to take after receiving a silence message is system // dependent. The semantics of the silence message in the Flash // Media playback and timing model are as follows: // // - Ensure all buffered audio data is played out before entering the // silence period: // Make sure that any audio data currently in the buffer is fully // processed and played. This ensures a clean transition into the // silence period without cutting off any audio. // // - After playing all buffered audio data, flush the audio decoder: // Clear the audio decoder to reset its state and prepare it for new // input after the silence period. // // - During the silence period, the audio clock can't be used as the // master clock for synchronizing playback: // Switch to using the system's wall-clock time to maintain the correct // timing for video and other data streams. // } </pre> |

```

audioPacketType = UB[4] as AudioPacketType // at byte boundary after this read

if (audioPacketModExType == AudioPacketModExType.TimestampOffsetNano) {
    // This block processes TimestampOffsetNano to enhance RTMP timescale
    // accuracy and compatibility with formats like MP4, M2TS, and Safari's
    // Media Source Extensions. It ensures precise synchronization without
    // altering core RTMP timestamps, applying only to the current media
    // message. These adjustments enhance synchronization and timing
    // accuracy in media messages while preserving the core RTMP timestamp
    // integrity.
    //
    // NOTE:
    // - 1 millisecond (ms) = 1,000,000 nanoseconds (ns).
    // - Maximum value representable with 20 bits is 1,048,575 ns
    //   (just over 1 ms), allowing precise sub-millisecond adjustments.
    // - modExData must be at least 3 bytes, storing values up to 999,999 ns.
    audioTimestampNanoOffset = bytesToUI24(modExData)

    // TODO: Integrate this nanosecond offset into timestamp management
    // to accurately adjust the presentation time.
}

if (audioPacketType == AudioPacketType.Multitrack) {
    isAudioMultitrack = true;
    audioMultitrackType = UB[4] as AvMultitrackType

    // Fetch AudioPacketType for all audio tracks in the audio message.
    // This fetch MUST not result in a AudioPacketType.Multitrack
    audioPacketType = UB[4] as AudioPacketType

    if (audioMultitrackType != AvMultitrackType.ManyTracksManyCodecs) {
        // The tracks are encoded with the same codec. Fetch the FOURCC for them
        audioFourCc = FOURCC as AudioFourCc
    } else {
        audioFourCc = FOURCC as AudioFourCc
    }
}

// - Don't wait for audio frames for synchronized A+V playback:
//   Normally, audio frames drive the synchronization of audio and video
//   (A/V) playback. During the silence period, playback should not stall
//   waiting for audio frames. Video and other data streams should
//   continue to play based on the wall-clock time, ensuring smooth
//   playback without audio.
//
// AudioPacketType.SequenceEnd is to have no less than the same meaning as
// a silence message. While it may seem redundant, we need to introduce
// this enum to ensure we can signal the end of the audio sequence for any
// audio track.
SequenceEnd      = 2,
// 3 - Reserved
MultichannelConfig = 4,
// Turns on audio multitrack mode
Multitrack       = 5,
// 6 - reserved
// ModEx is a special signal within the AudioPacketType enum that
// serves to both modify and extend the behavior of the current packet.
// When this signal is encountered, it indicates the presence of
// additional modifiers or extensions, requiring further processing to
// adjust or augment the packet's functionality. ModEx can be used to
// introduce new capabilities or modify existing ones, such as
// enabling support for high-precision timestamps or other advanced
// features that enhance the base packet structure.
ModEx           = 7,
// ...
// 14 - reserved
// 15 - reserved
}

enum AudioPacketModExType {
    TimestampOffsetNano = 0,
    //
    // ...
    // 14 - reserved
    // 15 - reserved
}

enum AudioFourCc {
    //
    // Valid FOURCC values for signaling support of audio codecs
    // in the enhanced FourCC pipeline. In this context, support
    // for a FourCC codec MUST be signaled via the enhanced
    // "connect" command.
    //
}

```

```

// AC-3/E-AC-3 - <https://en.wikipedia.org/wiki/Dolby_Digital>
Ac3      = makeFourCc("ac-3"),
Eac3     = makeFourCc("ec-3"),

// Opus audio - <https://opus-codec.org/>
Opus     = makeFourCc("Opus"),

// Mp3 audio - <https://en.wikipedia.org/wiki/MP3>
Mp3     = makeFourCc(".mp3"),

// Free Lossless Audio Codec - <https://xiph.org/flac/format.html>
Flac    = makeFourCc("fLaC"),

// Advanced Audio Coding - <https://en.wikipedia.org/wiki/Advanced_Audio_Coding>
// The following AAC profiles, denoted by their object types, are supported
// 1 = main profile
// 2 = low complexity, a.k.a., LC
// 5 = high efficiency / scale band replication, a.k.a., HE / SBR
Aac     = makeFourCc("mp4a"),
}

enum AvMultitrackType {
//
// Used by audio and video pipeline
//

OneTrack      = 0,
ManyTracks    = 1,
ManyTracksManyCodecs = 2,

//
// 3 - Reserved
// ...
// 15 - Reserved
}

```

ExAudioTagBody Section

Note: This ExAudioTagBody format is signaled by the presence of **ExAudioTagHeader**

| Description Of Bitstream | Enumerated Types |
|--|--|
| <pre> // // process ExAudioTagBody // while (processAudioBody) { if (isAudioMultitrack) { if (audioMultitrackType == AvMultitrackType.ManyTracksManyCodecs) { // Each track has a codec assigned to it. Fetch the FOURCC for the next track. audioFourCc = FOURCC as AudioFourCc } // Track Ordering: // // To provide a consistent convention, it is RECOMMENDED that trackId 0 be </pre> | <pre> enum AudioChannelOrder { // // Only the channel count is specified, without any further information // about the channel order // Unspecified = 0, // // The native channel order (i.e., the channels are in the same order in // which as defined in the AudioChannel enum). // Native = 1, </pre> |

```

// used for the default track of a given media type. The default track is
// the representation that the publisher expects most receivers to select
// when no additional selection logic is applied, for example the primary
// or most broadly applicable presentation for the stream.
//
// Additional variants, for example different bitrates, resolutions,
// codecs, languages, or camera angles, SHOULD use distinct positive
// trackId values (1, 2, 3, ...). These values are identifiers only and do
// not imply any inherent ordering, priority, or quality ranking.
//
// Encoders and ingesters SHOULD provide complete and accurate information
// in the onMetaData fields for each track. This includes codec identifiers
// and other descriptive track-level attributes such as bitrate, resolution,
// sample rate, channel count, language, and similar properties relevant to
// track selection and processing.
//
// The authoritative details for each track are also present within the
// stream itself, for example through codec configuration records and fully
// self-describing media packets. These in-stream signals can be used by
// receivers when determining appropriate processing behavior.
//
// The full definition of track selection or priority logic is beyond the
// scope of the E-RTMP specification. The guidance provided here is intended
// only to establish a consistent convention for trackId usage and related
// metadata structure. Individual implementations can use different
// approaches when combining onMetaData information with in-stream
// signaling to perform track selection or processing.
//
// Implementations MUST NOT infer detailed quality or compatibility
// characteristics from the trackId alone. Instead, they SHOULD consider
// the metadata provided via onMetaData together with the information
// conveyed within the media stream to evaluate characteristics such as
// bitrate, resolution, codec, language, or device profile alignment when
// choosing a track.
audioTrackId = UI8

if (audioMultitrackType != AvMultitrackType.OneTrack) {
    // The `sizeOfAudioTrack` specifies the size in bytes of the
    // current track that is being processed. This size starts
    // counting immediately after the position where the `sizeOfAudioTrack`
    // value is located. You can use this value as an offset to locate the
    // next audio track in a multitrack system. The data pointer is
    // positioned immediately after this field. Depending on the MultiTrack
    // type, the offset points to either a `fourCc` or a `trackId`.
    sizeOfAudioTrack = UI24
}

if (audioPacketType == AudioPacketType.MultichannelConfig) {
    //
    // Specify a speaker for a channel as it appears in the bitstream.
    // This is needed if the codec is not self-describing for channel mapping
}

```

```

//
// The channel order does not correspond to any predefined
// order and is stored as an explicit map.
//
Custom      = 2

// 3 - Reserved
// ...
// 15 - reserved
}

enum AudioChannelMask {
    //
    // Mask used to indicate which channels are present in the stream.
    //

    // masks for commonly used speaker configurations
    // <https://en.wikipedia.org/wiki/Surround_sound#Standard_speaker_channels>
    FrontLeft      = 0x000001,
    FrontRight     = 0x000002,
    FrontCenter    = 0x000004,
    LowFrequency1  = 0x000008,
    BackLeft       = 0x000010,
    BackRight      = 0x000020,
    FrontLeftCenter = 0x000040,
    FrontRightCenter = 0x000080,
    BackCenter     = 0x000100,
    SideLeft       = 0x000200,
    SideRight      = 0x000400,
    TopCenter      = 0x000800,
    TopFrontLeft   = 0x001000,
    TopFrontCenter = 0x002000,
    TopFrontRight  = 0x004000,
    TopBackLeft    = 0x008000,
    TopBackCenter  = 0x010000,
    TopBackRight   = 0x020000,
    //
    // Completes 22.2 multichannel audio, as
    // standardized in SMPTE ST2036-2-2008
    // see - <https://en.wikipedia.org/wiki/22.2_surround_sound>
    LowFrequency2  = 0x040000,
    TopSideLeft    = 0x080000,
    TopSideRight   = 0x100000,
    BottomFrontCenter = 0x200000,
    BottomFrontLeft = 0x400000,
    BottomFrontRight = 0x800000,
}

enum AudioChannel {
    //
    // Channel mappings enums
    //
}

```

```

// set audio channel order
audioChannelOrder = UI8 as AudioChannelOrder

// number of channels
channelCount = UI8

if (audioChannelOrder == AudioChannelOrder.Custom) {
    // Each entry specifies the speaker layout (see AudioChannel enum above
    // for layout definition) in the order that it appears in the bitstream.
    // First entry (i.e., index 0) specifies the speaker layout for channel 1.
    // Subsequent entries specify the speaker layout for the next channels
    // (e.g., second entry for channel 2, third entry for channel 3, etc.).
    audioChannelMapping = UI8[channelCount] as AudioChannel[]
}

if (audioChannelOrder == AudioChannelOrder.Native) {
    // audioChannelFlags indicates which channels are present in the
    // multi-channel stream. You can perform a Bitwise AND
    // (i.e., audioChannelFlags & AudioChannelMask.xxx) to see if a
    // specific audio channel is present
    audioChannelFlags = UI32
}

if (audioPacketType == AudioPacketType.SequenceEnd) {
    // signals end of sequence
}

if (audioPacketType == AudioPacketType.SequenceStart) {
    if (audioFourCc == AudioFourCc.Aac) {
        // The AAC audio specific config (a.k.a., AacSequenceHeader) is
        // defined in ISO/IEC 14496-3.
        aacHeader = [AacSequenceHeader]
    }

    if (audioFourCc == AudioFourCc.Flac) {
        // FlacSequenceHeader layout is:
        //
        // The bytes 0x66 0x4C 0x61 0x43 ("fLaC" in ASCII) signature
        //
        // Followed by a metadata block (called the STREAMINFO block) as described
        // in section 7 of the FLAC specification. The STREAMINFO block contains
        // information about the whole sequence, such as sample rate, number of
        // channels, total number of samples, etc. It MUST be present as the first
        // metadata block in the sequence. The FLAC audio specific bitstream format
        // is defined at <https://xiph.org/flac/format.html>
        flacHeader = [FlacSequenceHeader]
    }

    if (audioFourCc == AudioFourCc.Opus) {

```

```

        // commonly used speaker configurations
        // see - <https://en.wikipedia.org/wiki/Surround\_sound#Standard\_speaker\_channels>
        FrontLeft      = 0, // i.e., FrontLeft is assigned to channel zero
        FrontRight,
        FrontCenter,
        LowFrequency1,
        BackLeft,
        BackRight,
        FrontLeftCenter,
        FrontRightCenter,
        BackCenter      = 8,
        SideLeft,
        SideRight,
        TopCenter,
        TopFrontLeft,
        TopFrontCenter,
        TopFrontRight,
        TopBackLeft,
        TopBackCenter      = 16,
        TopBackRight,
        LowFrequency2      = 18,
        TopSideLeft,
        TopSideRight,
        BottomFrontCenter,
        BottomFrontLeft,
        BottomFrontRight      = 23,
        // 24 - Reserved
        // ...
        // 0xfd - reserved
        // Channel is empty and can be safely skipped.
        Unused      = 0xfe,
        // Channel contains data, but its speaker configuration is unknown.
        Unknown      = 0xff,
    }
}
```

```

// Opus Sequence header (a.k.a., ID header):
// - The Opus sequence start is also known as the ID header.
// - It contains essential information needed to initialize
// the decoder and understand the stream format.
// - For detailed structure, refer to RFC 7845, Section 5.1:
//   <https://datatracker.ietf.org/doc/html/rfc7845#section-5.1>
//
// If the Opus sequence start payload is empty, use the
// AudioPacketType.MultichannelConfig signal for channel
// mapping when present; otherwise, default to mono/stereo mode.
opusHeader = [OpusSequenceHeader]
}

}

if (audioPacketType == AudioPacketType.CodedFrames) {
if (audioFourCc == AudioFourCc.Ac3 || audioFourCc == AudioFourCc.Eac3) {
// Body contains audio data as defined by the bitstream syntax
// in the ATSC standard for Digital Audio Compression (AC-3, E-AC-3)
ac3Data = [Ac3CodedData]
}

if (audioFourCc == AudioFourCc.Opus) {
// Body contains Opus packets. The layout is one Opus
// packet for each of N different streams, where N is
// typically one for mono or stereo, but MAY be greater
// than one for multichannel audio. The value N is
// specified in the ID header (Opus sequence start) or
// via the AudioPacketType.MultichannelConfig signal, and
// is fixed over the entire length of the Opus sequence.
// The first (N - 1) Opus packets, if any, are packed one
// after another using the self-delimiting framing from
// Appendix B of [RFC6716]. The remaining Opus packet is
// packed at the end of the Ogg packet using the regular,
// undelimited framing from Section 3 of [RFC6716]. All
// of the Opus packets in a single audio packet MUST be
// constrained to have the same duration.
opusData = [OpusCodedData]
}

if (audioFourCc == AudioFourCc.Mp3) {
// An Mp3 audio stream is built up from a succession of smaller
// parts called frames. Each frame is a data block with its own header
// and audio information
mp3Data = [Mp3CodedData]
}

if (audioFourCc == AudioFourCc.Aac) {
// The AAC audio specific bitstream format is defined in ISO/IEC 14496-3.
aacData = [AacCodedData]
}

if (audioFourCc == AudioFourCc.Flac) {
}
}

```

```

// The audio data is composed of one or more audio frames. Each frame
// consists of a frame header, which contains a sync code and information
// about the frame, such as the block size, sample rate, number of
// channels, et cetera. The Flac audio specific bitstream format
// is defined at <https://xiph.org/flac/format.html>
flacData = [FlacCodedData]
}

}

if (
  isAudioMultitrack &&
  audioMultitrackType != AvMultitrackType.OneTrack &&
  positionDataPtrToNextAudioTrack(sizeOfAudioTrack)
) {
  // TODO: need to implement positionDataPtrToNextAudioTrack()
  continue
}

// done processing audio message
break
}

```

Enhanced Video

The **VideoTagHeader** has been extended to define additional video codecs, multitrack capabilities, signaling support, and additional miscellaneous enhancements, while ensuring backward compatibility. This extension is termed the **ExVideoTagHeader** and is designed to be future-proof, allowing for the definition of additional video codecs, features, and corresponding signaling.

During the parsing process, the logic MUST handle unexpected or unknown elements gracefully. Specifically, if any critical signaling or flags (e.g., **VideoFrameType**, **VideoPacketType**, or **VideoFourCc**) are not recognized, the system MUST fail in a controlled and predictable manner.

^ ^ >

Important: A single video message for a unique timestamp may include a batch of **VideoPacketType** values (e.g., multiple **trackId** values, **Metadata** values). When parsing a video message, the bitstream MUST be processed completely to ensure all payload data has been handled.

^ ^ >

Table: Extended VideoTagHeader

| Description Of Bitstream | Enumerated Types |
|--|---|
| <pre> // Check if isExVideoHeader flag is set to 1, signaling enhanced RTMP // video mode. In this case, VideoCodecId's 4-bit unsigned binary (UB[4]) // should not be interpreted as a codec identifier. Instead, these // UB[4] bits should be interpreted as VideoPacketType. isExVideoHeader = Boolean(UB[1]) videoFrameType = UB[3] as VideoFrameType if (isExVideoHeader == 0) { </pre> | <pre> enum VideoFrameType { // 0 - reserved KeyFrame = 1, // a seekable frame InterFrame = 2, // a non - seekable frame DisposableInterFrame = 3, // H.263 only GeneratedKeyFrame = 4, // reserved for server use only } // If videoFrameType is not ignored and is set to VideoFrameType.Command, </pre> |

```

// Utilize the VideoCodecId values and the bitstream description
// as defined in the legacy [FLV] specification. Refer to this
// version for the proper implementation details.
videoCodecId = UB[4] as VideoCodecId

if (videoFrameType == VideoFrameType.Command) {
    videoCommand = UI8 as VideoCommand
}
}

// the payload will not contain video data. Instead, (Ex)VideoTagHeader
// will be followed by a UI8, representing the following meanings:
//
//      0 = Start of client-side seeking video frame sequence
//      1 = End of client-side seeking video frame sequence
//
// frameType is ignored if videoPacketType is VideoPacketType.MetaData
Command          = 5,      // video info / command frame

// 6 = reserved
// 7 = reserved
}

enum VideoCommand {
    StartSeek = 0,
    EndSeek   = 1,

    // 0x03 = reserved
    // ...
    // 0xff = reserved
}

enum VideoCodecId {
    // These values remain as they were in the legacy [FLV] specification.
    // If the IsExVideoHeader flag is set, we switch into
    // FOURCC video mode defined in the VideoFourCc enumeration.
    // This means that VideoCodecId (UB[4] bits) is not interpreted
    // as a codec identifier. Instead, these UB[4] bits are
    // interpreted as VideoPacketType.

    // 0 - Reserved
    // 1 - Reserved
    SorensonH263   = 2,
    Screen         = 3,
    On2VP6         = 4,
    On2VP6A        = 5, // with alpha channel
    ScreenV2       = 6,
    Avc            = 7,
    // 8 - Reserved
    // ...
    // 15 - Reserved
}

```

ExVideoTagHeader Section
note: **ExVideoTagHeader** is present if **IsExVideoHeader** flag is set.

| Description Of Bitstream | Enumerated Types |
|--|---|
| <pre> // process ExVideoTagHeader // processVideoBody = false if (isExVideoHeader) { </pre> | <pre> enum VideoPacketType { SequenceStart = 0, CodedFrames = 1, SequenceEnd = 2, </pre> |

```

processVideoBody = true

// Interpret UB[4] bits as VideoPacketType instead of sound rate, size, and type.
videoPacketType = UB[4] as VideoPacketType // at byte boundary after this read

// Process each ModEx data packet
while (videoPacketType == VideoPacketType.ModEx) {
    // Determine the size of the packet ModEx data (ranging from 1 to 256 bytes)
    modExDataSize = UI8 + 1

    // If maximum 8-bit size is not sufficient, use a 16-bit value
    if (modExDataSize == 256) {
        modExDataSize = UI16 + 1;
    }

    // Fetch the packet ModEx data based on its determined size
    modExData = UI8[modExDataSize]

    // fetch the VideoPacketOptionType
    videoPacketModExType = UB[4] as VideoPacketModExType

    // Update videoPacketType
    videoPacketType = UB[4] as VideoPacketType // at byte boundary after this read

    if (videoPacketModExType == VideoPacketModExType.TimestampOffsetNano) {
        // This block processes TimestampOffsetNano to enhance RTMP timescale
        // accuracy and compatibility with formats like MP4, M2TS, and Safari's
        // Media Source Extensions. It ensures precise synchronization without
        // altering core RTMP timestamps, applying only to the current media
        // message. These adjustments enhance synchronization and timing
        // accuracy in media messages while preserving the core RTMP timestamp
        // integrity.
        //
        // NOTE:
        // - 1 millisecond (ms) = 1,000,000 nanoseconds (ns).
        // - Maximum value representable with 20 bits is 1,048,575 ns
        //   (just over 1 ms), allowing precise sub-millisecond adjustments.
        // - modExData must be at least 3 bytes, storing values up to 999,999 ns.
        videoTimestampNanoOffset = bytesToUI24(modExData)

        // TODO: Integrate this nanosecond offset into timestamp management
        // to accurately adjust the presentation time.
    }
}

if (
    videoPacketType != VideoPacketType.Metadata &&
    videoFrameType == VideoFrameType.Command
) {
    videoCommand = UI8 as VideoCommand

    // ExVideoTagBody has no payload if we got here.
}

```

```

// CompositionTime Offset is implicitly set to zero. This optimization
// avoids transmitting an SI24 composition time value of zero over the wire.
// See the ExVideoTagBody section below for corresponding pseudocode.
CodedFramesX = 3,

// ExVideoTagBody does not contain video data. Instead, it contains
// an AMF-encoded metadata. Refer to the Metadata Frame section for
// an illustration of its usage. For example, the metadata might include
// HDR information. This also enables future possibilities for expressing
// additional metadata meant for subsequent video sequences.
//
// If VideoPacketType.Metadata is present, the FrameType flags
// at the top of this table should be ignored.
Metadata = 4,

// Carriage of bitstream in MPEG-2 TS format
//
// PacketTypeSequenceStart and PacketTypeMPEG2TSSequenceStart
// are mutually exclusive
MPEG2TSSequenceStart = 5,

// Turns on video multitrack mode
Multitrack = 6,

// ModEx is a special signal within the VideoPacketType enum that
// serves to both modify and extend the behavior of the current packet.
// When this signal is encountered, it indicates the presence of
// additional modifiers or extensions, requiring further processing to
// adjust or augment the packet's functionality. ModEx can be used to
// introduce new capabilities or modify existing ones, such as
// enabling support for high-precision timestamps or other advanced
// features that enhance the base packet structure.
ModEx = 7,

// 8 - Reserved
// ...
// 14 - reserved
// 15 - reserved
}

enum VideoPacketModExType {
    TimestampOffsetNano = 0,
    //
    // ...
    // 14 - reserved
    // 15 - reserved
}

enum VideoFourCc {
    //
    // Valid FOURCC values for signaling support of video codecs
}

```

```

// Set boolean to not try to process the video body.
processVideoBody = false
} else if (videoPacketType == VideoPacketType.Multitrack) {
isVideoMultitrack = true;
videoMultitrackType = UB[4] as AvMultitrackType

// Fetch VideoPacketType for all video tracks in the video message.
// This fetch MUST not result in a VideoPacketType.Multitrack
videoPacketType = UB[4] as VideoPacketType

if (videoMultitrackType != AvMultitrackType.ManyTracksManyCodecs) {
    // The tracks are encoded with the same codec. Fetch the FOURCC for them
    videoFourCc = FOURCC as VideoFourCc
}
} else {
    videoFourCc = FOURCC as VideoFourCc
}
}

// in the enhanced FourCC pipeline. In this context, support
// for a FourCC codec MUST be signaled via the enhanced
// "connect" command.
//
Vp8      = makeFourCc("vp08"),
Vp9      = makeFourCc("vp09"),
Av1      = makeFourCc("av01"),
Avc      = makeFourCc("avc1"),
Hevc     = makeFourCc("hvc1"),
}

enum AvMultitrackType {
    //
    // Used by audio and video pipeline
    //

    OneTrack      = 0,
    ManyTracks    = 1,
    ManyTracksManyCodecs = 2,
    //
    // 3 - Reserved
    //
    // ...
    //
    // 15 - Reserved
}

```

ExVideoTagBody Section

Note: This **ExVideoTagBody** format is signaled by the presence of **ExVideoTagHeader** and if **videoCommand** has not been set (see [VideoFrameType](#) description)

Description Of Bitstream

```

// process ExVideoTagBody
//
while (processVideoBody) {
    if (isVideoMultitrack) {
        if (videoMultitrackType == AvMultitrackType.ManyTracksManyCodecs) {
            // Each track has a codec assigned to it. Fetch the FOURCC for the next track.
            videoFourCc = FOURCC as VideoFourCc
        }

        // Track Ordering:
        //
        // To provide a consistent convention, it is RECOMMENDED that trackId 0 be
        // used for the default track of a given media type. The default track is
        // the representation that the publisher expects most receivers to select
        // when no additional selection logic is applied, for example the primary
        // or most broadly applicable presentation for the stream.
        //
        // Additional variants, for example different bitrates, resolutions,
        // codecs, languages, or camera angles, SHOULD use distinct positive
        // trackId values (1, 2, 3, ...). These values are identifiers only and do
        // not imply any inherent ordering, priority, or quality ranking.
        //
    }
}

```

```

// Encoders and ingesters SHOULD provide complete and accurate information
// in the onMetaData fields for each track. This includes codec identifiers
// and other descriptive track-level attributes such as bitrate, resolution,
// sample rate, channel count, language, and similar properties relevant to
// track selection and processing.
//
// The authoritative details for each track are also present within the
// stream itself, for example through codec configuration records and fully
// self-describing media packets. These in-stream signals can be used by
// receivers when determining appropriate processing behavior.
//
// The full definition of track selection or priority logic is beyond the
// scope of the E-RTMP specification. The guidance provided here is intended
// only to establish a consistent convention for trackId usage and related
// metadata structure. Individual implementations can use different
// approaches when combining onMetaData information with in-stream
// signaling to perform track selection or processing.
//
// Implementations MUST NOT infer detailed quality or compatibility
// characteristics from the trackId alone. Instead, they SHOULD consider
// the metadata provided via onMetaData together with the information
// conveyed within the media stream to evaluate characteristics such as
// bitrate, resolution, codec, language, or device profile alignment when
// choosing a track.
videoTrackId = UI8

if (videoMultitrackType != AvMultitrackType.OneTrack) {
    // The `sizeOfVideoTrack` specifies the size in bytes of the
    // current track that is being processed. This size starts
    // counting immediately after the position where the `sizeOfVideoTrack`
    // value is located. You can use this value as an offset to locate the
    // next video track in a multitrack system. The data pointer is
    // positioned immediately after this field. Depending on the MultiTrack
    // type, the offset points to either a `fourCc` or a `trackId.`
    sizeOfVideoTrack = UI24
}
}

if (videoPacketType == VideoPacketType.Metadata) {
    // The body does not contain video data; instead, it consists of AMF-encoded
    // metadata. The metadata is represented by a series of [name, value] pairs.
    // Currently, the only defined [name, value] pair is ["colorInfo", Object].
    // See the Metadata Frame section for more details on this object.
    //
    // For a deeper understanding of the encoding, please refer to the descriptions
    // of SCRIPTDATA and SCRIPTDATAVALUE in the FLV file specification.
    videoMetadata = [VideoMetadata]
}

if (videoPacketType == VideoPacketType.SequenceEnd) {
    // signals end of sequence
}

```

```

if (videoPacketType == VideoPacketType.SequenceStart) {
    if (videoFourCc == VideoFourCc.Vp8) {
        // body contains a VP8 configuration record to start the sequence
        vp8Header = [VPCodecConfigurationRecord]
    }

    if (videoFourCc == VideoFourCc.Vp9) {
        // body contains a VP9 configuration record to start the sequence
        vp9Header = [VPCodecConfigurationRecord]
    }

    if (videoFourCc == VideoFourCc.Av1) {
        // body contains a configuration record to start the sequence
        av1Header = [AV1CodecConfigurationRecord]
    }

    if (videoFourCc == VideoFourCc.Avc) {
        // body contains a configuration record to start the sequence.
        // See ISO/IEC 14496-15:2019, 5.3.4.1 for the description of
        // the AVCDriverConfigurationRecord.
        avcHeader = [AVCDriverConfigurationRecord]
    }

    if (videoFourCc == VideoFourCc.Hevc) {
        // body contains a configuration record to start the sequence.
        // See ISO/IEC 14496-15:2022, 8.3.3.2 for the description of
        // the HEVCDecoderConfigurationRecord.
        hevcHeader = [HEVCDecoderConfigurationRecord]
    }
}

if (videoPacketType == VideoPacketType.MPEG2TSSequenceStart) {
    if (videoFourCc == VideoFourCc.Av1) {
        // body contains a video descriptor to start the sequence
        av1Header = [AV1VideoDescriptor]
    }
}

if (videoPacketType == VideoPacketType.CodedFrames) {
    if (videoFourCc == VideoFourCc.Vp8) {
        // body contains series of coded full frames
        vp8CodedData = [Vp8CodedData]
    }

    if (videoFourCc == VideoFourCc.Vp9) {
        // body contains series of coded full frames
        vp9CodedData = [Vp9CodedData]
    }

    if (videoFourCc == VideoFourCc.Av1) {
        // body contains one or more OBUs representing a single temporal unit
    }
}

```

```

    av1CodedData = [Av1CodedData]
}

if (videoFourCc == VideoFourCc.Avc) {
    // See ISO/IEC 14496-12:2015, 8.6.1 for the description of the composition
    // time offset. The offset in an FLV file is always in milliseconds.
    compositionTimeOffset = SI24

    // Body contains one or more NALUs; full frames are required
    avcCodedData = [AvcCodedData]
}

if (videoFourCc == VideoFourCc.Hevc) {
    // See ISO/IEC 14496-12:2015, 8.6.1 for the description of the composition
    // time offset. The offset in an FLV file is always in milliseconds.
    compositionTimeOffset = SI24

    // Body contains one or more NALUs; full frames are required
    hevcData = [HevcCodedData]
}

if (videoPacketType == VideoPacketType.CodedFramesX) {
    // compositionTimeOffset is implied to equal zero. This is
    // an optimization to save putting SI24 value on the wire

    if (videoFourCc == VideoFourCc.Avc) {
        // Body contains one or more NALUs; full frames are required
        avcCodedData = [AvcCodedData]
    }

    if (videoFourCc == VideoFourCc.Hevc) {
        // Body contains one or more NALUs; full frames are required
        hevcData = [HevcCodedData]
    }
}

if (
    isVideoMultitrack &&
    videoMultitrackType != AvMultitrackType.OneTrack &&
    positionDataPtrToNextVideoTrack(sizeOfVideoTrack)
) {
    // TODO: need to implement positionDataPtrToNextVideoTrack()
    continue
}

// done processing video message
break
}

```

Metadata Frame

To support various types of video metadata, the legacy [\[FLV\]](#) specification has been enhanced. The VideoTagHeader has been extended to define a new **VideoPacketType.Metadata** (see ExVideoTagHeader table in [Enhanced Video](#) section) whose payload will contain an AMF-encoded metadata. The metadata will be represented by a series of [name, value] pairs. For now the only defined [name, value] pair is **["colorInfo", Object]**. When leveraging **VideoPacketType.Metadata** to deliver HDR metadata, the metadata MUST be sent prior to the video sequence, scene, frame or such that it affects. Each time a new **colorInfo** object is received it invalidates and replaces the current one. To reset to the original color state you can send **colorInfo** with a value of Undefined (the RECOMMENDED approach) or an empty object (i.e., {}).

It is intentional to leverage a video message to deliver **VideoPacketType.Metadata** instead of other [\[RTMP\]](#) Message types. One benefit of leveraging a video message is to avoid any racing conditions between video messages and other RTMP message types. Given this, once your **colorInfo** object is parsed, the read values MUST be processed in time to affect the first frame of the video section which follows the **colorInfo** object.

The **colorInfo** object provides HDR metadata to enable a higher quality image source conforming to BT.2020 (a.k.a., Rec. 2020) standard. The properties of the **colorInfo** object, which are encoded in an AMF message format, are defined below.

^^>

Note:

- For content creators: Whenever it behooves to add video hint information via metadata (e.g., HDR) to the FLV container it is RECOMMENDED to add it via **VideoPacketType.Metadata**. This may be done in addition (or instead) to encoding the metadata directly into the codec bitstream.
- The object encoding format (i.e., AMF0 or AMF3) is signaled during the [connect](#) command.

^^>

```js

```
type ColorInfo = {
 colorConfig: {
 // number of bits used to record the color channels for each pixel
 bitDepth: number, // SHOULD be 8, 10 or 12

 //
 // colorPrimaries, transferCharacteristics and matrixCoefficients are defined
 // in ISO/IEC 23091-4/ITU-T H.273. The values are an index into
 // respective tables which are described in "Colour primaries",
 // "Transfer characteristics" and "Matrix coefficients" sections.
 // It is RECOMMENDED to provide these values.
 //

 // indicates the chromaticity coordinates of the source color primaries
 colorPrimaries: number, // enumeration [0-255]

 // opto-electronic transfer characteristic function (e.g., PQ, HLG)
 transferCharacteristics: number, // enumeration [0-255]

 // matrix coefficients used in deriving luma and chroma signals
 matrixCoefficients: number, // enumeration [0-255]
 },
}
```

```

hdrCll: {
 //
 // maximum value of the frame average light level
 // (in 1 cd/m2) of the entire playback sequence
 //
 maxFall: number, // [0.0001-10000]

 //
 // maximum light level of any single pixel (in 1 cd/m2)
 // of the entire playback sequence
 //
 maxCLL: number, // [0.0001-10000]
}

//
// The hdrMdcv object defines mastering display (i.e., where
// creative work is done during the mastering process) color volume (a.k.a., mdcv)
// metadata which describes primaries, white point and min/max luminance. The
// hdrMdcv object SHOULD be provided.
//
// Specification of the metadata along with its ranges adhere to the
// ST 2086:2018 - SMPTE Standard (except for minLuminance see
// comments below)
//
hdrMdcv: {
 //
 // Mastering display color volume (mdcv) xy Chromaticity Coordinates within CIE
 // 1931 color space.
 //
 // Values SHALL be specified with four decimal places. The x coordinate SHALL
 // be in the range [0.0001, 0.7400]. The y coordinate SHALL be
 // in the range [0.0001, 0.8400].
 //
 redX: number,
 redY: number,
 greenX: number,
 greenY: number,
 blueX: number,
 blueY: number,
 whitePointX: number,
 whitePointY: number,

 //
 // max/min display luminance of the mastering display (in 1 cd/m2 ie. nits)
 //
 // note: ST 2086:2018 - SMPTE Standard specifies minimum display mastering
 // luminance in multiples of 0.0001 cd/m2.
 //
 // For consistency we specify all values
 // in 1 cd/m2. Given that a hypothetical perfect screen has a peak brightness
 // of 10,000 nits and a black level of .0005 nits we do not need to
}

```

```

// switch units to 0.0001 cd/m2 to increase resolution on the lower end of the
// minLuminance property. The ranges (in nits) mentioned below suffice
// the theoretical limit for Mastering Reference Displays and adhere to the
// SMPTE ST 2084 standard (a.k.a., PQ) which is capable of representing full gamut
// of luminance level.
//
maxLuminance: number, // [5-10000]
minLuminance: number, // [0.0001-5]
},
}

```

**Table: Flag values for the `videoFunction` property**

| Function Flag                                 | Usage                                                                                                                                                                                                                                                                                   | Value  |
|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|
| SUPPORT_VID_CLIENT_SEEK                       | Indicates that the client can perform frame-accurate seeks.                                                                                                                                                                                                                             | 0x0001 |
| SUPPORT_VID_CLIENT_HDR                        | Indicates that the client has support for HDR video. Note: Implies support for <code>colorInfo</code> Object within <code>VideoPacketType.Metadata</code> .                                                                                                                             | 0x0002 |
| SUPPORT_VID_CLIENT_VIDEO_PACKET_TYPE_METADATA | Indicates that the client has support for <code>VideoPacketType.Metadata</code> . See Metadata Frame section for more detail.                                                                                                                                                           | 0x0004 |
| SUPPORT_VID_CLIENT_LARGE_SCALE_TILE           | The large-scale tile allows the decoder to extract only an interesting section in a frame without the need to decompress the entire frame. Support for this feature is not required and is assumed to not be implemented by the client unless this property is present and set to true. | 0x0008 |

## Multitrack Streaming via Enhanced RTMP

### Introduction to Multitrack Capabilities

E-RTMP has introduced support for multitrack streaming, offering increased flexibility in audio and video streaming through the use of a track index (i.e., `audioTrackId` and `videoTrackId`). This feature allows for the serialization of multiple tracks over a single E-RTMP connection and stream channel.

It's important to note that multitrack support is designed to augment, not replace, the option of using multiple streams for streaming. While both multiple streams and multitrack can potentially address the same use cases, the choice between them will depend on the specific capabilities of your E-RTMP implementation and requirements. In certain cases, multitrack may not be the most efficient option.

### Multitrack Sample Use Cases

- **Adaptive Bitrate Streaming:** Multitrack support allows the client to send Adaptive Bitrate (ABR) ladders, thus avoiding the need for server-side transcoding and reducing quality loss. This also facilitates sending content with multiple codecs like AV1, HEVC, and VP9.
- **Device Specific Streaming:** The feature allows for the streaming of different aspect ratios, tailored for various device profiles, enabling more dynamic and flexible presentations.

- **Frame-Level Synchronization:** For example, you can synchronize multiple camera views in a concert.
- **Multi-Language Support:** Support for multiple audio tracks in a single [[FLV](#)] file is now available, eliminating the need for multiple file versions.
- **Multi-codec publishing:** Allows a publisher to encode media in multiple formats within a single stream and enabling delivery of the appropriate codec based on platform support

## Multitrack Media Message Guidelines

- **Video Messages:** Each video message MUST include a **videoTrackId** (refer to the **videoPacketType.Multitrack** entry in the **ExVideoTagHeader** table within the [Enhanced Video](#) section for video bitstream signaling) as it is not persistent across messages.
- **Audio Messages:** Similarly, each audio message MUST include an **audioTrackId** (refer to the **AudioPacketType.Multitrack** in the **ExAudioTagHeader** table within the [Enhanced Audio](#) section for audio bitstream signaling).
- **Payload Parsing:** All tracks within a single timestamp MUST be processed to ensure comprehensive media handling.
- **Track Ordering:** To provide a consistent convention, it is RECOMMENDED that trackId 0 be used for the default track of a given media type. The default track is the representation that the publisher expects most receivers to select when no additional selection logic is applied, for example the primary or most broadly applicable presentation for the stream.

Additional variants, for example different bitrates, resolutions, codecs, languages, or camera angles, SHOULD use distinct positive trackId values (1, 2, 3, ...). These values are identifiers only and do not imply any inherent ordering, priority, or quality ranking.

Encoders and ingesters SHOULD provide complete and accurate information in the **onMetaData** fields for each track. This includes codec identifiers and other descriptive track-level attributes such as bitrate, resolution, sample rate, channel count, language, and similar properties relevant to track selection and processing.

The authoritative details for each track are also present within the stream itself, for example through codec configuration records, and fully self-describing media packets. These in-stream signals can be used by receivers when determining the appropriate processing behavior.

The full definition of track selection or priority logic is beyond the scope of the E-RTMP specification. The guidance provided here is intended only to establish a consistent convention for trackId usage and related metadata structure. Individual implementations can use different approaches when combining **onMetaData** information with in-stream signaling to perform track selection or processing.

Implementations MUST NOT infer detailed quality or compatibility characteristics from the trackId alone. Instead, they SHOULD consider the metadata provided via **onMetaData** together with the information conveyed within the media stream to evaluate characteristics such as bitrate, resolution, codec, language, or device profile alignment when choosing a track.

## SCRIPTDATA Multitrack Parameter Handling

- **trackId SHOULD be a Parameter:** For methods within RTMP that involve [[SCRIPTDATA](#)] messages, the **trackId** can be a critical parameter for operations that pertain to specific media tracks. In such cases, the **trackId** SHOULD be passed in as an argument to the method, ensuring

that the action or data manipulation is accurately applied to the correct track. When track namespaces include both audio and video tracks, the combination of `trackId` and `mediaType` uniquely identifies the target track.

- **Recommended Parameter Passing:**

- **Using a Map or Object Argument:** The recommended way to pass the `trackId` to methods involving `SCRIPTDATA` is by including it within a map or as a property of an object argument. This approach aligns with practices such as those used in the [Enhancing onMetaDate](#) section, enhancing consistency and scalability across various implementations.
- **Function Signature Example:** This method takes an object arg which contains the properties `trackId` and `mediaType`. This structure is effective for managing multiple parameters and enhances readability and maintainability.

```
```js
// Invokes a media-type specific script for a given track.
function scriptMethodName(arg: {
  trackId: number,
  mediaType: "video" | "audio" | "data",
  // ... additional properties can be added here
}) {
  console.log(`Invoking ${arg.mediaType} script for trackId:`, arg.trackId);
}
```
```

```

Note: This example illustrates one possible object structure. Actual `SCRIPTDATA` payloads may vary depending on the encoder or tooling, and may use maps or other AMF object forms consistent with historical `SCRIPTDATA` usage.

- **Advantages of Parameter Passing Approach:**

- **Clarity and Structure:** Using an object or map to pass arguments, including the `trackId`, organizes the parameters neatly and reduces the chances of errors or misalignment in parameter order.
- **Enhanced Maintenance:** It becomes easier to add more parameters in the future without altering the method signature drastically, thereby maintaining compatibility and easing future enhancements. The style of passing the `trackId` as part of a structured object or map ensures a robust framework for handling `SCRIPTDATA` operations in RTMP streams, providing clear, scalable, and error-free management of track-specific data.

Leveraging Multitrack Features in E-RTMP

Multitrack capabilities in E-RTMP offer a wide range of possibilities, from adaptive bitrate streaming to multi-language support. While this document doesn't prescribe specific encoding rules or manifest metadata, it aims to guide you through the complexities of leveraging multitrack features. Consider various parameters like codecs, frame rates, key frames, sampling rates, and resolutions to meet your unique objectives. Remember, media encoding settings are separate from E-RTMP configurations.

Enhancing NetConnection connect Command

When a client connects to an E-RTMP server, it sends a [connect](#) command to the server. The command structure sent from the client to the server contains a Command Object, comprising name-value pairs. This is where the client indicates the audio and video codecs it supports. To declare support for newly defined codecs or other enhancements supported by the client, this name-value pair list must be extended. Below is the description of a new name-value pair used in the Command Object of the [connect](#) command.

Table: New name-value pair that can be set in the Command Object

Property	Type	Description	Example Value
fourCcList	Strict Array of strings	<p>Used to declare the enhanced list of supported codecs when connecting to the server. The fourCcList property is a strict array of dense ordinal indices. Each entry in the array is of string type, specifically a [FourCC] value (i.e., a string that is a sequence of four bytes), representing a supported audio/video codec.</p> <p>In the context of E-RTMP, clients capable of receiving any codec (e.g., recorders or forwarders) may set a FourCC value to the wildcard value of "*".</p> <p>Note: The fourCcList property was introduced in the original E-RTMP. Going forward, it is RECOMMENDED on the client side to switch to using the [audio video]FourCcInfoMap properties described below. On the server side, we RECOMMEND supporting both fourCcList and [audio video]FourCcInfoMap properties to handle cases where a client has not yet transitioned to using the new properties.</p>	<pre>e.g., 1 ["av01", "vp09", "vp08", "Hvc1", "Avc1", "ac-3", "ec-3", "Opus", ".mp3", "fLaC", "Aac"]</pre> <pre>e.g., 2 ["*"]</pre>
videoFourCcInfoMap, audioFourCcInfoMap	Object	<p>The [audio video]FourCcInfoMap properties are designed to enable setting capability flags for each supported codec in the context of E-RTMP streaming. A FourCC key is a four-character code used to specify a video or audio codec. The names of the object properties are strings that correspond to these FourCC keys. Each object property holds a numeric value that represents a set of capability flags. These flags can be combined using a Bitwise OR operation.</p> <p>Refer to the enum FourCcInfoMask for the available flags:</p> <pre>enum FourCcInfoMask { CanDecode = 0x01, CanEncode = 0x02, CanForward = 0x04, }</pre> <p>Capability flags define specific functionalities, such as the ability to decode, encode, or forward.</p> <p>A FourCC key set to the wildcard character "*" acts as a catch-all for any codec. When this wildcard key exists, it overrides the flags set on properties for specific codecs. For example, if the flag for the "*" property is set to FourCcInfoMask.CanForward, all</p>	<pre>e.g., 1 videoFourCcInfoMap = { // can forward any video codec "*": FourCcInfoMask.CanForward, // can decode, encode, forward (see "*") VP9 codec "vp09": FourCcInfoMask.CanDecode FourCcInfoMask.CanEncode, }</pre> <pre>e.g., 2 audioFourCcInfoMap = { // can forward any audio codec "*": FourCcInfoMask.CanForward, // can decode, encode, forward (see "*") Opus codec "Opus": FourCcInfoMask.CanDecode FourCcInfoMask.CanEncode, }</pre>

		codecs will be forwarded regardless of individual flags set on their specific properties.	
capsEx	number	<p>The value represents capability flags which can be combined via a Bitwise OR to indicate which extended set of capabilities (i.e., beyond the legacy [RTMP] specification) are supported via E-RTMP. See enum <code>CapsExMask</code> for the enumerated values representing the assigned bits. If the extended capabilities are expressed elsewhere they will not appear here (e.g., FourCC, HDR or <code>VideoPacketType.Metadata</code> support is not expressed in this property).</p> <p>When a specific flag is encountered:</p> <ul style="list-style-type: none"> - The implementation might fully handle the feature by applying the appropriate logic. - Alternatively, if full support is not available, the implementation can still parse the bitstream correctly, ensuring graceful degradation. This allows continued operation, even with reduced functionality. <pre>enum CapsExMask { Reconnect = 0x01, // Support for reconnection Multitrack = 0x02, // Support for multitrack ModEx = 0x04, // Can parse ModEx signal TimestampNanoOffset = 0x08, // Support for nano offset }</pre>	CapsExMask.Reconnect CapsExMask.Multitrack

As you can see, the client declares to the server what enhancements it supports. The server responds with a command, either `_result` or `_error`, to indicate whether the response is a result or an error. During the response, the server provides some properties within an Object as one of the parameters. This is where the server needs to state its support for E-RTMP. The server SHOULD state its support via attributes such as `videoFourCcInfoMap`, `capsEx`, and similar properties.

Action Message Format (AMF): AMF0 and AMF3

Action Message Format (AMF) is a compact binary format used to serialize `SCRIPTDATA`. It has two specifications: [AMF0] and [AMF3]. AMF3 improves on AMF0 by optimizing the payload size on the wire. To understand the full scope of these optimizations, please refer to the AMF0 and AMF3 specifications.

Supporting AMF3 in the [RTMP] and [FLV] is beneficial due to its optimization over AMF0. Understanding the ecosystem is crucial before adding AMF3 support to RTMP or FLV.

Enabling AMF3 in RTMP

To enable support for AMF3 in RTMP, the following steps are REQUIRED:

- Adding support for [Data Message](#), [Shared Object Message](#) and [Command Message](#) and their associated AMF3 message types (i.e., [15](#), [16](#) and [17](#)).
- Adding support for the AMF3 set of possible type markers ([see AMF3 specification section 3.1](#)).
- Signaling in the [connect](#) command that the AMF3 encoding format is supported in addition to AMF0.

RTMP has had AMF3 as part of its specification for some time now. During the handshake, the client declares whether it has support for AMF3.

Enabling AMF3 in FLV

Prior to Y2023, the FLV file format did not have AMF3 as part of its **SCRIPTDATA** specification. To ensure support for AMF3 in FLV:

- Add a new FLV **TagType** 15 (i.e., in addition to **TagType** 18), which supports **SCRIPTDATA** encoded via AMF3 (i.e., similar to the way Data Message is handled).

Important AMF3-encoded Historical Specification Clarification

Established, pre E-RTMP, specifications state the following:

- Command Messages carry the AMF-encoded commands between the client and the server. Message type values:
 - 20 for AMF0 encoding.
 - 17 for AMF3 encoding.
- Data Messages are sent by the client or server to send Metadata or user data to the peer, including details such as creation time, duration, theme, etc. Message type values:
 - 18 for AMF0 encoding.
 - 15 for AMF3 encoding.
- The message types 19 for AMF0 and 16 for AMF3 are reserved for Shared Object events.
- AMF0 was extended to allow an AMF0 encoding context to be switched to AMF3. A new type marker, [avmplus-object-marker \(byte 0x11\)](#), was added. The presence of this marker signifies that the following value is encoded in AMF3. Legacy AMF0 systems that haven't been updated to support AMF3 should throw an unknown type error.

Unfortunately, the above is incomplete and may be somewhat unclear. To clarify, in addition to the above:

- Object Encoding property in the Command Object of the **connect** command indicates the type of serialization (a.k.a., encoding) supported by the client or server:
 - A value of 0 (default and optional) indicates support for AMF0 encoding and message types of 18, 19 and 20.
 - A value of 3 indicates support for both AMF0 and AMF3 encoding and message types of (18, 15), (19, 16) and (20, 17).
- Message payload for message types of 15, 16 and 17 starts with a format selector byte. Currently, only format 0 is defined to indicate AMF0-encoded values. It's possible to signal a switch to AMF3 serialization by prefixing an AMF3 value with an AMF0 [avmplus-object-marker \(byte 0x11\)](#). The switch isn't sticky, and parsing MUST return to AMF0 encoding mode once the AMF3 value is serialized. This means that every AMF3 encoded value MUST be prefixed with an [avmplus-object-marker \(byte 0x11\)](#) as defined in AMF0.

Protocol Versioning

There is no need for a version bump within E-RTMP for either the [RTMP] handshake sequence or the FLV header file version field. All of the enhancements are triggered via the newly defined additions to the bitstream format which don't break legacy implementations. E-RTMP is self describing in its capabilities.

References

[AMF0]

Adobe Systems Inc. "Action Message Format – AMF 0", June 2006,
[<https://veovera.github.io/enhanced-rtmp/docs/legacy/amf0-file-format-spec.pdf>](https://veovera.github.io/enhanced-rtmp/docs/legacy/amf0-file-format-spec.pdf).

[AMF3]

Adobe Systems Inc. "Action Message Format – AMF 3", June 2006,
[<https://veovera.github.io/enhanced-rtmp/docs/legacy/amf3-file-format-spec.pdf>](https://veovera.github.io/enhanced-rtmp/docs/legacy/amf3-file-format-spec.pdf).

[DEPRECATED]

Deprecation,
[<https://en.wikipedia.org/wiki/Deprecation>](https://en.wikipedia.org/wiki/Deprecation).

[FLV]

"Adobe Flash Video File Format Specification, Version 10.1", August 2010,
[<https://veovera.github.io/enhanced-rtmp/docs/legacy/video-file-format-v10-1-spec.pdf>](https://veovera.github.io/enhanced-rtmp/docs/legacy/video-file-format-v10-1-spec.pdf).

[FourCC]

A sequence of four bytes (typically ASCII) used to uniquely identify data formats,
[<https://en.wikipedia.org/wiki/FourCC>](https://en.wikipedia.org/wiki/FourCC).

[LEGACY]

Legacy specifications for the RTMP solution,
<<https://veovera.github.io/enhanced-rtmp/docs/legacy/>>.

[RFC2119]

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997,
<<https://www.rfc-editor.org/info/rfc2119>>.

[RFC8174]

Leiba, B., "Ambiguity of Uppercase vs Lowercase in [RFC 2119](#) Key Words", [BCP 14](#), [RFC 8174](#), DOI 10.17487/RFC8174, May 2017,
<<https://www.rfc-editor.org/info/rfc8174>>.

[RTMP]

Parmar, H., Ed. and M. Thornburgh, Ed., "Adobe's Real Time Messaging Protocol", December 2012,
<<https://veovera.github.io/enhanced-rtmp/docs/legacy/rtmp-v1-0-spec.pdf>>.

[ScriptTagBody]

"Adobe Flash Video File Format Specification, Version 10.1", August 2010,
<<https://veovera.org/docs/legacy/video-file-format-v10-1-spec.pdf#page=80>>.

[SCRIPTDATA]

"Adobe Flash Video File Format Specification, Version 10.1", August 2010,
<<https://veovera.org/docs/legacy/video-file-format-v10-1-spec.pdf#page=80>>.

[WebCodecs]

W3C, "WebCodecs"
<<https://www.w3.org/TR/webcodecs/>>.

