

Enhanced RTMP

Table of Contents

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

Document Status

Author: Slavik Lozben (Veovera Software Organization)

Contributors: Adobe, Google, Twitch, Jean-Baptiste Kempf (FFmpeg, VideoLAN), pkv (OBS), Dennis Sädler (OBS), Xavier Hallade (Intel Corporation), Luxoft, SplitmediaLabs Limited (XSplit), Craig Barberich (VSO), Michael Thornburgh

Status: **v2-2024-04-02-a1**

Alpha Version Disclaimer for Enhanced RTMP V2 Specification

This document outlines the alpha version of the Enhanced Real-Time Messaging Protocol (RTMP) Version 2 specification. As we refine and enhance the protocol, there may be breaking changes introduced to this alpha version based on feedback and further testing. Rest assured, no breaking changes will occur in the General Availability (GA) released versions.

We encourage developers, implementers, and stakeholders to actively participate in this development phase. Your feedback, whether it be bug reports, feature suggestions, or usability improvements, is invaluable and can be submitted via new issues in our GitHub repository at <https://github.com/veovera/enhanced-rtmp>. We are committed to transparently communicating updates and changes, ensuring that all stakeholders are informed and involved.

Please note that engaging with this alpha version gives you a unique opportunity to influence the final specifications of Enhanced RTMP V2. We look forward to collaborating with you on this exciting journey towards a more robust and efficient protocol.

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 (VP8, VP9, HEVC, AV1) and audio codecs (AC3, EAC3, Opus, FLAC). 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:

- Integration of advanced video codecs (VP8, VP9, HEVC, AV1) with High Dynamic Range (HDR) support, enhancing video quality for modern displays.
- Introduction of VideoPacketType.Metadata, broadening the scope of video metadata.
- Implementation of video multitrack capabilities, facilitating sophisticated media stream management.
- Establishment of a reconnect request feature, bolstering connection stability and resilience.

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. 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, fostering a vibrant ecosystem around enhanced RTMP.

Our commitment to RTMP's evolution doesn't stop here. We plan to quickly follow up these enhancements with the incorporation of leading audio codecs, such as Opus, FLAC, AC-3, and E-AC-3, along with the development of features supporting audio multitrack and multichannel configurations. This continuous innovation underscores our dedication to keeping RTMP at the forefront of streaming technology standards.

Introduction

This document describes enhancements to [\[RTMP\]](#) and [\[FLV\]](#), 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 enhanced RTMP specification has been crafted and should be interpreted.

- **Enhanced RTMP:** refers to a series of improvements made to the legacy Real-Time Messaging Protocol (RTMP), originally developed by Adobe. It's important to note that **enhanced RTMP** is not a brand name but a term used to distinguish this advanced version from the legacy [\[RTMP\]](#) specification. Endorsed by Adobe and widely adopted across the industry, enhanced RTMP serves as the current standard for RTMP solutions. This updated protocol includes various enhancements to both RTMP and the FLV format. Please be aware that the term **enhanced RTMP** signifies ongoing updates to RTMP and FLV, and does not pertain to any specific iteration or release.
- **Pseudocode:** Pseudocode has been provided to convey logic on how to interpret the RTMP or FLV 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 are used to define valid values
 - 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.
- Enhanced 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 enhanced 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 enhanced 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 enhanced 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 enhanced 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 enhanced 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 enhanced 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.

Contextualizing Enhancements

In the following section, we'll outline key enhancements. The aim is to give readers a clear snapshot of the enhanced RTMP objectives and intentions before diving into the rest of the detailed specifications.

- Newly introduced codecs

Table: Additional codecs for RTMP

Additional Video Codec	Notes
AVC (a.k.a., H.264, added FOURCC signaling)	Popular within streaming hardware and software solutions.
HEVC (a.k.a., H.265)	
VP8 (webRTC officially supports this codec)	
VP9	
AV1	<ul style="list-style-type: none">• Gaining popularity• Codec agnostic services are asking for AV1 support

- HDR - to accommodate new video codecs and cater to the existing spectrum of displays
- VideoPacketType.Metadata - to accommodate diverse video metadata types
- Multitrack - to provide the ability to manage or process multiple video tracks
- and more...

Simple Data Types

The following data types are used in RTMP bitstreams and FLV files. FOURCC was introduced to support enhanced RTMP and FLV.

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

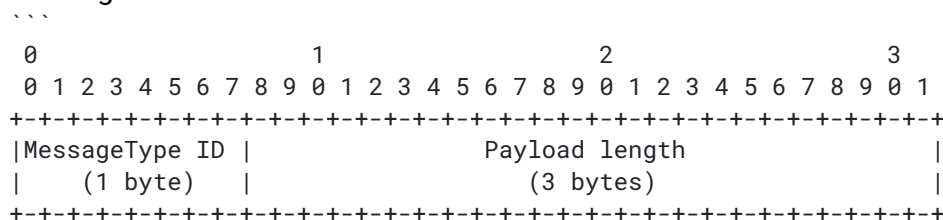
Note: Unless specifically called out, multi-byte integers SHALL be stored in big-endian byte order

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



		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 pre 2023 (a.k.a., 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
AVCPacketType	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)

		IF AVCPacketType == 1
		Composition time offset
CompositionTime	IF CodecID == 7	ELSE
	SI24	0
		See ISO 14496-12, 8.15.3 for an explanation of composition times. The offset in an FLV file is always in milliseconds.

Enhancements to RTMP and FLV

Within the following sections, this document provides a comprehensive overview of the enhancements made to RTMP and FLV. These improvements 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	Audio codec ID used in the file: See AudioTagHeader of the legacy [FLV] specification for available CodecID values.
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	Resolution of a single 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	Video codec ID used in the file: See VideoTagHeader of the legacy [FLV] specification for available CodecID values. 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).
videodatarate	number	Video bitrate, in kilobits per second
width	number	Width of the video, in pixels

Note: The property **videocodecid** has been enhanced to support FOURCC (Four-byte ASCII code) values. These values are interpreted as UI32 (e.g., 'av01').

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 Video

The VideoTagHeader has been extended to define additional video codecs, multitrack capabilities, additional miscellaneous enhancements, and signaling support, 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 = UB[1] videoFrameType = UB[3] as VideoFrameType if (isExVideoHeader == 0) { // Utilize the VideoCodecId values and the bitstream description</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, // the payload will not contain video data. Instead, (Ex)VideoTagHeader</pre>

```

// as defined in the legacy [ELV] specification. Refer to this
// version for the proper implementation details.
videoCodecId = UB[4] as VideoCodecId

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

// 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 [ELV] 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

```

//
// process ExVideoTagHeader
//
processVideoBody = false
if (isExVideoHeader == 1) {
    processVideoBody = true

    // The UB[4] bits are interpreted as VideoPacketType
    // instead of VideoCodecId
    videoPacketType = UB[4] as VideoPacketType

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

```

Enumerated Types

```

enum VideoPacketType {
    SequenceStart = 0,
    CodedFrames = 1,
    SequenceEnd = 2,

    // 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.
}

```



```

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

```

```

//
// 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,

// 7 - Reserved
// ...
// 14 - reserved
// 15 - reserved
}

enum VideoFourCc {
//
// Valid FOURCC values for signaling support of video codecs
// in the enhanced FourCC pipeline. In this context, support
// for a FourCC codec MUST be signaled via the enhanced
// 'connect' command.
//
Av1 = makeFourCc('av01'),
Vp8 = makeFourCc('vp08'),
Vp9 = makeFourCc('vp09'),
Hevc = makeFourCc('hvc1'),
Avc = makeFourCc('avc1'),
}

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:
//
// For identifying the highest priority (a.k.a., default track)
// or highest quality track, it is RECOMMENDED to use trackId
// set to zero. For tracks of lesser priority or quality, use
// multiple instances of trackId with ascending numerical values.
// The concept of priority or quality can have multiple
// interpretations, including but not limited to bitrate,
// resolution, default angle, and language. This recommendation
// serves as a guideline intended to standardize track numbering
// across various applications.
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

```

```

// 14496-15, 5.2.4.1 for the description of AVCDecoderConfigurationRecord
avcHeader = [AVCDecoderConfigurationRecord]
}

if (videoFourCc == VideoFourCc.Hevc) {
// body contains a configuration record to start the sequence. See ISO
// ISO 14496-15, 8.3.3.1.2 for the description of 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 which represent a single temporal unit
av1CodedData = [Av1CodedData]
}

if (videoFourCc == VideoFourCc.Avc) {
// See ISO 14496-12, 8.15.3 for an explanation of composition times.
// 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 14496-12, 8.15.3 for an explanation of composition times.
// 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.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)) {
    // positionDataPtrToNextVideoTrack() is for developer to write
    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 PacketTypeMetadata 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 PacketTypeMetadata 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 (ex. 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 (ex. PQ, HLG)
 transferCharacteristics: number, // enumeration [0-255]

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

 hdrCl1: {
 //
 // 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 colorInfo Object within VideoPacketType.Metadata.	0x0002
SUPPORT_VID_CLIENT_VIDEO_PACKET_TYPE_METADATA	Indicates that the client has support for VideoPacketType.Metadata. 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

Enhanced RTMP has introduced support for multitrack streaming, offering increased flexibility in audio and video streaming through the use of a track index (a.k.a., **trackId**). This feature allows for the serialization of multiple tracks over a single 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 RTMP implementation and requirements. In certain cases, multitrack may not be the most efficient option.

## Sample Multitrack 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.

## Additional Multitrack Details

- **Video Messages:** Each video message should include a **trackId** (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.
- **Payload Parsing:** All tracks within a single timestamp must be processed to ensure comprehensive media handling.
- **Track Ordering:** For identifying the highest priority (a.k.a., default track) or highest quality track, it is RECOMMENDED to use **trackId** set to zero. For tracks of lesser priority or quality, use multiple instances of **trackId** with ascending numerical values. The concept of **priority** or **quality** can have multiple interpretations, including but not limited to bitrate, resolution, default angle, and language. This recommendation serves as a guideline intended to standardize track numbering across various applications.

## General Guidelines

Multitrack capabilities in enhanced 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 Enhanced RTMP configurations.

## Enhancing NetConnection connect Command

When a client connects to an 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 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 <b>fourCcList</b> property is a strict array of dense ordinal indices. Each entry in the array is of string type, specifically a <b>FourCC</b> value (i.e., a string that is a sequence of four bytes), representing a supported video codec.</p> <p>In the context of enhanced RTMP, clients capable of receiving any codec (e.g., recorders or forwarders) may set a <b>FourCC</b> value to the wildcard value of '*'.</p> <p>Note: The <b>fourCcList</b> property was introduced in the original enhanced RTMP. Going forward, it is RECOMMENDED on the client side to switch to using the <b>videoFourCcInfoMap</b> properties described below. On the server side, we RECOMMEND supporting both <b>fourCcList</b> and <b>videoFourCcInfoMap</b> properties to handle cases where a client has not yet transitioned to using the new properties.</p>	<p>e.g., 1 [ 'av01', 'vp09', 'hvc1', 'Avc1' ]</p> <p>e.g., 2 [ * ]</p>
videoFourCcInfoMap	Object	<p>The <b>videoFourCcInfoMap</b> property is designed to enable setting capability flags for each supported codec in the context of enhanced RTMP streaming. A <b>FourCC</b> key is a four-character code used to specify a video codec. The names of the object properties are strings that correspond to these <b>FourCC</b> 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 <b>FourCcInfoMask</b> 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 <b>FourCC</b> 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 <b>FourCcInfoMask.CanForward</b>, all codecs will be forwarded regardless of individual flags set on their specific properties.</p>	<p>e.g., 1</p> <pre>videoFourCcInfoMap = {     // can forward any video codec     '*': FourCcInfoMask.CanForward,      // can decode, encode, forward (see '*') VP9 codec     'vp09': FourCcInfoMask.CanDecode               FourCcInfoMask.CanEncode, }</pre>
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 <a href="#">RTMP</a> specification) are supported via enhanced</p>	<p>CapsExMask.Reconnect   CapsExMask.Multitrack</p>



RTMP. See enum CapsExMask 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 VideoPacketType.Metadata support is not expressed in this property).

```
enum CapsExMask {
 Reconnect = 0x01 // See reconnect section
 Multitrack = 0x02, // See multitrack section
}
```

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 enhanced 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-enhanced 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 enhanced 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. Enhanced RTMP is self describing in its capabilities.

## Documentation Versioning

### Overview

This section outlines our standardized approach for versioning our specification documentation. Effective versioning ensures consistency, enables users to identify the latest version easily, and facilitates collaboration among team members.

### File Naming Convention

We name the documentation files with a clear identifier and the major version number.

Example:

**enhanced-rtmp-v2.pdf**

### Version Information Inside the Document

We include a dedicated section or metadata within each document to specify the version details which includes the major version number, date, and stage of development (alpha/beta/release).

Example:

**Status: v2-2024-02-26-a1**

### Calendar Versioning Format Description

The format for versioning documents is structured as follows:

- **v#-yyy-mm-dd-[a|b|r]#:**
  - **v#:** Major version number for tracking the progression of the enhanced RTMP development.

- **yyyy-mm-dd**: Date when the document was updated.
- **[a|b|r]**: Suffix to distinguish between the alpha, beta, and release stage.
- **#**: Minor version number for a particular date. Increments for multiple versions on the same date.

This format provides a comprehensive overview of each version's status and chronological order, facilitating effective tracking and management of the enhanced RTMP specification development.

## 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>>.

### [AMF3]

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

### [DEPRECATED]

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>>.

### [FourCC]

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>>.

## Appendix

---

### Document Revision History

Document Revision History	
Date	Comments
v2-2024-03-16-a1	1. The Enhanced RTMP Version 2 Alpha is now ready for public testing.
v2-2024-04-02-a1	1. Fixed pseudocode logic relating to VP8 sequence start and coded data.