Enhancing RTMP, FLV

With Additional Video Codecs And HDR Support

Status: 2023-03-v1.0.0-B.5
Author: Slavik Lozben (VSO)
Contributors: Google, Jean-Baptiste Kempf (FFmpeg, VideoLAN), Luxoft, Dennis Sädtler (OBS), PKV (OBS), SplitmediaLabs Limited (XSplit), Xavier Hallade (Intel Corporation), Craig Barberich (VSO)

Document Revision History	
Date	Comments
Mar 22, 2023	1. Initial beta submission for Enhanced RTMP (note: Beta 1-4 are not in github repository).

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 pasted from [RFC 2119].

- **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 key word **DEPRECATED** to the set of key words above. We use the Wikipedia description for the key word.

- 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

There are ongoing requests from the media streaming industry to enhance the RTMP/FLV solution by bringing the protocol up to date with the current state of the art streaming technologies. RTMP was released over 20 years ago (The first public release of RTMP was in 2002). Many streaming solutions use RTMP in their stack today. While RTMP has remained popular it has gone a bit stale in its evolution. As an example RTMP/FLV does not have support for popular video codecs like VP9, HEVC, AV1. This document outlines enhancements to the RTMP/FLV specification to help bring this protocol inline with the current streaming media technologies by adding new capabilities to it. The new capabilities are outlined here in this spec. The additional capabilities added to this enhancement spec for the RTMP solution are:

- New video codec types:

Additional Video Codecs	Notes
HEVC (H.265)	Popular within streaming hardware and software solutions.
VP9	
AV1	• Gaining popularity • Codec agnostic services are asking for AV1 support

- HDR capability - To support new video codecs and the current range of displays
- PacketTypeMetadata - to support various types of video metadata

Introduction

This document describes enhancements to the RTMP spec by adding support for additional media codecs and HDR capability. One of the key goals is to ensure that this enhancement does not define any breaking changes to legacy clients and the content that they stream. This means that legacy RTMP/FLV specifications and documentations continue to stay valid and important to the RTMP ecosystem. This enhancement specification tries to limit duplication of information from legacy specifications. Legacy specification plus this documentation here form one holistic information for the RTMP solution. Some of the legacy informative references that have been leveraged are:

- Adobe RTMP Specification
- Adobe Flash Video File Format Specification Version 10.1
- Additional Useful Reading

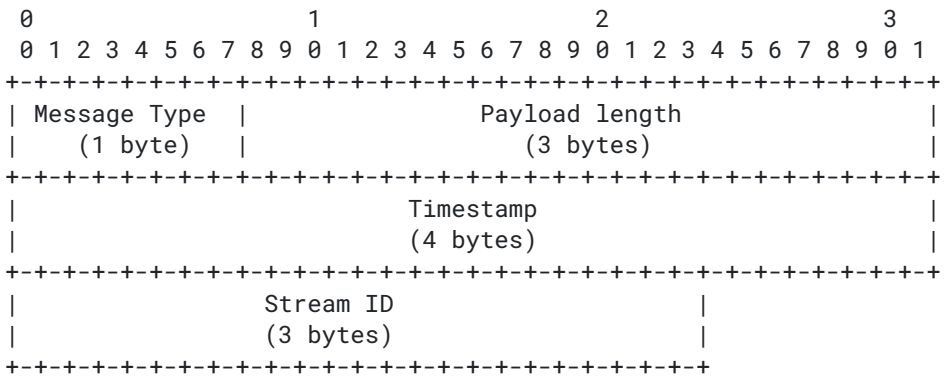
The Enhancement Spec Usage License

RTMP Message Format

Adobe's Real Time Messaging Protocol (RTMP) describes RTMP as "an application-level protocol designed for multiplexing and packetizing multimedia transport streams (such as audio, video, and interactive content) over a suitable transport protocol (such as TCP)". One of the most important features of RTMP is the Chunk Stream. Chunk Stream multiplexes, packetizes and prioritizes messages

on the wire. Chunking and prioritizing messages is the “RT” (i.e. Real Time) within RTMP. RTMP Message has two sections. A message **header** followed by a **message payload**:

- The format of the message header is describe below



  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

- Message payload follows the header. The payload for example could contain compressed audio data or compressed video data. RTMP knows nothing about the payload, including how to process it. If we want to add new codec types they have to be defined where the actual payload internals are defined. Flash Video (FLV) is a container file format where AV payload internals, including the codecs, are defined.
- Please see the original RTMP (in various locations) & FLV (in Annex E. on page 68) specification for the endianness (aka byte order) of the data format on the wire.

A TidBit About FLV File Format

FLV file is a container for AV data. The file consists of alternating back-pointers and tags. Each tag is accompanied by data related to that tag. Each TagType within an FLV file is defined by 5 bits. AUDIODATA has a tag type of 8 and VIDEODATA has a tag type of 9. Note: these Tag Types map to the same Message Type IDs in the RTMP spec. This is by design. Each Tag Types of 8 or 9 is accompanied by an AudioTagHeader or VideoTagHeader. It’s common to think of RTMP in conjunction with FLV. That said, RTMP is a protocol and FLV is a file container. This is why they are originally defined in separate specifications. This enhancement spec is enhancing both RTMP and FLV.

Current VideoTagHeader

Below is the VideoTagHeader format for the pre 2023 FLV spec (i.e. ver 10.1.2.01):

Table 2: Current 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)
CompositionTime	IF CodecID == 7 SI24	IF AVCPacketType == 1 Composition time offset ELSE 0 See ISO 14496-12, 8.15.3 for an explanation of composition times. The offset in an FLV file is always in milliseconds.

Note: We have 4 bits to define video CodecID, luckily not all values are taken. We will leverage available values to define additional video CodecIDs. Whew, we have room to define new video formats. We can leverage the unused bits to achieve unlimited codec expansion. Please see the **ENHANCEMENT TO RTMP & FLV** sections below for the enhancement descriptions.

ENHANCEMENT TO RTMP & FLV ARE DESCRIBED BELOW

Defining Additional Video Codecs

Below VideoTagHeader is extended to define additional video codecs and the supporting signaling while keeping backwards compatibility intact. The ExVideoTagHeader is future proof for defining additional codecs and the accompanying signaling. During parsing, logic **must** gracefully fail if at any point important signaling/flags (ex. FrameType, IsExHeader, ExHeaderInfo) are not understood.

Table 4: Extended VideoTagHeader

Field	Type	Comment

IsExHeader | FrameType	**UB[4]**	**IF (UB[4] & 0b1000) != 0 {** **IsExHeader = true** **// Signals to not interpret CodecID UB[4] as a codec identifier. Instead** **// these UB[4] bits are interpreted as PacketType which is then followed** **// by UI32 FourCC value.** **} ELSE {** **IsExHeader = true** **// Use CodecID values as described** **// in the pre 2023 FLV spec (i.e. ver 10.1.2.01):** **}** **// see ExVideoTagHeader section for description of PacketType** **IF PacketType != PacketTypeMetaData {** **// signal the type of video frame.** **FrameType = (UB[4] & 0b0111).** **}** The following FrameType values are defined: 0 = reserved 1 = key frame (a seekable frame) 2 = inter frame (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 6 = reserved 7 = reserved IF FrameType == 5, the payload will not contain video data. The VideoTagHeader will be followed by a UI8 and have the following meaning: - 0 = Start of client-side seeking video frame sequence - 1 = End of client-side seeking video frame sequence **Note: Backwards compatibility is preserved since the IsExHeader bit was always part of FrameType UB[4] but never defined/used. Pre 2023 FrameType values never reached 8 and the IsExHeader flag (aka most significant bit of FrameType UB[4]) was always zero in pre Y2023 specs).**
CodecID	IF IsExHeader == 0 UB[4]	Codec Identifier. The following values are defined: 0 = Reserved 1 = Reserved 2 = Sorenson H.263 3 = Screen video 4 = On2 VP6 5 = On2 VP6 with alpha channel 6 = Screen video version 2 7 = AVC 8 = Reserved 9 = Reserved 10 = Reserved 11 = Reserved 12 = Reserved 13 = Reserved 14 = Reserved 15 = Reserved **Note: Values remain as before (i.e. no changes made). Please note if the IsExHeader flag is set (see above) we switch into FourCC video mode defined below. That means that CodecId UB[4] bits are not interpreted as a codec identifier. Instead these UB[4] bits are interpreted as a PacketType. Another**

		way of stating this: the UB[4] bits are either CodecID or PacketType as part of the ExHeaderInfo. This is signaled by the IsExHeader flag.
ExVideoTagHeader Description Below note: ExVideoTagHeader header is present IF IsExHeader flag is set.		

PacketType
(i.e. not CodecId)
	IF IsExHeader == 1
	UB[4]
		0 = PacketTypeSequenceStart
		1 = PacketTypeCodedFrames
		2 = PacketTypeSequenceEnd
		
		// CompositionTime Offset is implied to equal zero. This is
		// an optimization to save putting SI24 composition time value of zero on
		// the wire. See pseudo code below in the VideoTagBody section
		3 = PacketTypeCodedFramesX
		
		// VideoTagBody does not contain video data. VideoTagBody
		// instead contains an AMF encoded metadata. See Metadata Frame
		// section for an illustration of its usage. As an example, the metadata
		// can be HDR information. This is a good way to signal HDR
		// information. This also opens up future ways to express additional
		// metadata that is meant for the next video sequence.
		//
		// note: presence of PacketTypeMetadata means that FrameType
		// flags at the top of this table should be ignored
		4 = PacketTypeMetadata
		
		// Carriage of bitstream in MPEG-2 TS format
		5 = PacketTypeMPEG2TSSequenceStart
		
		6 = Reserved
		…
		14 = reserved
		15 = reserved

Video FourCC	UI32	The following are the currently defined FourCC values to signal video codecs. AV1 = { 'a', 'v', '0', '1' } VP9 = { 'v', 'p', '0', '9' } HEVC = { 'h', 'v', 'c', '1' }
VideoTagBody Description Below		

```
IF PacketType == PacketTypeMetadata {
    // The body does not contain video data. The body is 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]
    // See Metadata Frame section for more details of this object.
    //
    // For a deeper understanding of the encoding please see description
    // of SCRIPTDATA and SSCRIPTDATAVALUE in the FLV file spec.
    DATA = [“colorInfo”, Object]
} ELSE IF PacketType == PacketTypeSequenceEnd {
    // signals end of sequence
}

IF FourCC == AV1 {
    IF PacketType == PacketTypeSequenceStart {
        // body contains a configuration record to start the sequence
        DATA = [AV1CodecConfigurationRecord]
    } ELSE IF PacketType == PacketTypeMPEG2TSSequenceStart {
        DATA = [AV1VideoDescriptor]
    } ELSE IF PacketType == PacketTypeCodedFrames {
        // body contains one or more OBUs which MUST represent a single temporal unit
```

```
        DATA = [Series of coded frames]
    }
}

If FourCC == VP9 {
    IF PacketType == PacketTypeSequenceStart {
        // body contains a configuration record to start the sequence
        DATA = [VPCodecConfigurationRecord]
    } ELSE IF PacketType == PacketTypeCodedFrames {
        // body MUST contain full frames
        DATA = [Series of coded frames]
    }
}
If FourCC == HEVC {
    IF PacketType == PacketTypeSequenceStart {
        // body contains a configuration record to start the sequence
        // See ISO 14496-15, 8.3.3.1.2 for the description of HEVCDecoderConfigurationRecord
        DATA = [HEVCDecoderConfigurationRecord]
    } ELSE IF PacketType == PacketTypeCodedFrames || PacketType == PacketTypeCodedFramesX {
        IF PacketType == PacketTypeCodedFrames {
            // See ISO 14496-12, 8.15.3 for an explanation of composition times.
            // The offset in an FLV file is always in milliseconds.
            SI24 = [CompositionTime Offset]
        } ELSE {
            // CompositionTime Offset is implied to equal zero. This is
            // an optimization to save putting SI24 value on the wire
        }
        // Body contains one or more NALUs; full frames are required
        DATA = [HEVC NALU]
    }
}
```

Extending 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. The Command Object is made up of name-value pairs. This is where the client indicates what audio and video codecs it supports. This name-value pair list will need to be extended to declare newly defined codecs, or any other enhancements, that are supported by the client. Following is the description of a new name-value pair used in Command Object of the connect command.

Table 5: new name-value pair that can be set in the Command Object

Property	Type	Description	Example Value
fourCcList	**Strict Array of Strings**	**The enhanced list of supported codecs. It's a strict array of dense ordinal indices. Each entry in the array is of String Type. Each entry is set to a fourCC value (i.e. a string that is a sequence of four bytes) representing a supported video codec.**	**["av01", "vp09", "hvc1"]**

Metadata Frame

To support various types of video metadata, the FLV container specification has been enhanced. The VideoTagHeader has been extended to define a new **PacketTypeMetadata** (see Table 4) 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 should be sent prior to the video sequence that it affects.

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 keyframe of the video sequence which follows the **colorInfo** object.

The **colorInfo** object provides HDR metadata to enable a higher quality image source conforming to BT.2020 (aka. 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 **PacketTypeMetadata.** 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.

```
//
// HDR metadata information which intended to be surfaced in the
// protocol/container outside the encoded bit stream.
//
// Note: Not all properties are guaranteed to be present when the colorInfo object
// is serialized. Presence of Serialized properties will depend on how the original
// content was mastered, encoded and intent
//
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]
},

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 (aka 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 (aka PQ) which is capable of representing full gamut
        // of luminance level.
        //
        maxLuminance:   Number,     // [5-10000]
        minLuminance:   Number,     // [0.0001-5]
    }
};
```

Table 6: 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 PacketTypeMetadata.**	**0x0002**
SUPPORT_VID_CLIENT_PACKET_TYPE_METADATA	**Indicates that the client has support for PacketTypeMetadata. 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**
Above values can be combined via logical OR		

TABLE 7: NormaAction Message Format (AMF0 vs AMF3)

Action Message Format (AMF) is a compact binary format that is used to serialize script data. AMF has two versions: AMF 0 [AMF0] and AMF 3 [AMF3]. One way AMF3 improves on AMF0 is by optimizing the payload size on the wire. To understand the

full scope of the optimizations please see the links above. It is RECOMMENDED to support AMF3 in the RTMP solution. It is nice to have AMF3 support within the FLV file. You should understand the ecosystem before adding AMF3 data to your FLV files.

The way to insure support for AMF3 in RTMP is by:

- Adding support for Command Message, Data Message and Shared Object Message and types which use AMF3 form.
- Signaling in the connect command that the object encoding format is AMF3.

Note: 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.

The way to insure support for AMF3 in FLV is by:

- Adding a new TagType 15 (i.e., not 18) which supports SCRIPTDATA which is encoded via AMF3 (i.e., similar to the way Data Message is handled)

Note: Prior to Y2023 FLV file format did not have AMF3 as part of its SCRIPTDATA specification.

Versioning

There is no need for a version bump in the RTMP handshake sequence or in the FLV header file version field. All of the extensions are based on already predefined format bits. The client/server logic will trigger based on the format bits and not on the versioning of the RTMP handshake or FLV header file version field.