

6MAN
Internet-Draft
Obsoletes: 6874 (if approved)
Updates: 3986, 3987 (if approved)
Intended status: Standards Track
Expires: 12 August 2022

B. Carpenter
Univ. of Auckland
S. Cheshire
Apple Inc.
R. Hinden
Check Point Software
8 February 2022

Representing IPv6 Zone Identifiers in Address Literals and Uniform
Resource Identifiers
draft-carpen-6man-rfc6874bis-03

Abstract

This document describes how the zone identifier of an IPv6 scoped address, defined as <zone_id> in the IPv6 Scoped Address Architecture (RFC 4007), can be represented in a literal IPv6 address and in a Uniform Resource Identifier that includes such a literal address. It updates the URI Generic Syntax and Internationalized Resource Identifier specifications (RFC 3986, RFC 3987) accordingly, and obsoletes RFC 6874.

Discussion Venue

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the 6MAN mailing list (ipv6@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/browse/ipv6/> (<https://mailarchive.ietf.org/arch/browse/ipv6/>).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 12 August 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
2. Issues with Implementing RFC 6874	4
3. Specification	4
4. URI Parsers	6
5. Security Considerations	7
6. Acknowledgements	7
7. References	7
7.1. Normative References	7
7.2. Informative References	8
Appendix A. Options Considered	9
Appendix B. Change log	10
Authors' Addresses	11

1. Introduction

The Uniform Resource Identifier (URI) syntax specification [RFC3986] defined how a literal IPv6 address can be represented in the "host" part of a URI. Two months later, the IPv6 Scoped Address Architecture specification [RFC4007] extended the text representation of limited-scope IPv6 addresses such that a zone identifier may be concatenated to a literal address, for purposes described in that specification. Zone identifiers are especially useful in contexts in which literal addresses are typically used, for example, during fault diagnosis, when it may be essential to specify which interface is used for sending to a link-local address. It should be noted that zone identifiers have purely local meaning within the node in which they are defined, often being the same as IPv6 interface names. They are completely meaningless for any other node. Today, they are meaningful only when attached to addresses with less than global scope, but it is possible that other uses might be defined in the future.

The IPv6 Scoped Address Architecture specification [RFC4007] does not specify how zone identifiers are to be represented in URIs.

Practical experience has shown that this feature is useful or necessary, in at least three use cases:

1. When using a web browser for simple debugging actions involving link-local addresses on a host with more than one active link interface.
2. When using a web browser to configure or reconfigure a device which only has a link local address and whose only configuration tool is a web server, again from a host with more than one active link interface.
3. When using an HTTP-based protocol for establishing link-local relationships, such as the Apple CUPS printing mechanism [CUPS].

It should be noted that whereas some operating systems and network APIs support a default zone identifier as described in [RFC4007], others do not, and for them an appropriate URI syntax is particularly important.

In the past, some browser versions directly accepted the IPv6 Scoped Address syntax [RFC4007] for scoped IPv6 addresses embedded in URIs, i.e., they were coded to interpret a "%" sign following the literal address as introducing a zone identifier [RFC4007], instead of introducing two hexadecimal characters representing some percent-encoded octet [RFC3986]. Clearly, interpreting the "%" sign as introducing a zone identifier is very convenient for users, although it is not supported by the URI syntax [RFC3986] or the Internationalized Resource Identifier (IRI) syntax [RFC3987]. Therefore, this document updates RFC 3986 and RFC 3987 by adding syntax to allow a zone identifier to be included in a literal IPv6 address within a URI.

It should be noted that in contexts other than a user interface, a zone identifier is mapped into a numeric zone index or interface number. The MIB textual convention InetZoneIndex [RFC4001] and the socket interface [RFC3493] define this as a 32-bit unsigned integer. The mapping between the human-readable zone identifier string and the numeric value is a host-specific function that varies between operating systems. The present document is concerned only with the human-readable string.

Several alternative solutions were considered while this document was developed. Appendix A briefly describes the various options and their advantages and disadvantages.

This document obsoletes its predecessor [RFC6874] by greatly simplifying its recommendations and requirements for URI parsers. Its effect on the formal URI syntax [RFC3986] is different from that of RFC 6874.

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.

2. Issues with Implementing RFC 6874

Several issues prevented RFC 6874 being implemented in browsers:

1. There was some disagreement with requiring percent-encoding of the "%" sign preceding a zone identifier. This requirement is dropped in the present document.
2. The requirement to delete any zone identifier before emitting a URI from the host in an HTTP message was considered both too complex to implement and in violation of normal HTTP practice [RFC7230]. This requirement has been dropped from the present document.
3. The suggestion to pragmatically allow a bare "%" sign when this would be unambiguous was considered both too complex to implement and confusing for users. This suggestion has been dropped from the present document since it is now irrelevant.

3. Specification

According to IPv6 Scoped Address syntax [RFC4007], a zone identifier is attached to the textual representation of an IPv6 address by concatenating "%" followed by <zone_id>, where <zone_id> is a string identifying the zone of the address. However, the IPv6 Scoped Address Architecture specification gives no precise definition of the character set allowed in <zone_id>. There are no rules or de facto standards for this. For example, the first Ethernet interface in a host might be called %0, %1, %en1, %eth0, or whatever the implementer happened to choose. Also, %25 would be valid.

In a URI, a literal IPv6 address is always embedded between "[" and "]". This document specifies how a <zone_id> can be appended to the address. According to the text in Section 2.4 of [RFC3986], "%" must be percent-encoded as "%25" to be used as data within a URI. However, in the formal ABNF syntax of RFC 3986, this only applies where the "pct-encoded" element appears. For this reason, it is

possible to extend the ABNF such that the scoped address `fe80::abcd%en1` would appear in a URI as `http://[fe80::abcd%en1]` or `https://[fe80::abcd%en1]`.

A `<zone_id>` MUST contain only ASCII characters classified as "unreserved" for use in URIs [RFC3986]. This excludes characters such as `]` or even `%` that would complicate parsing. The `<zone_id>` `"25"` cannot be forbidden since it is valid in some operating systems, so a parser MUST NOT apply percent decoding to a URI such as `http://[fe80::abcd%25]`.

If an operating system uses any other characters in zone or interface identifiers that are not in the "unreserved" character set, they cannot be used in a URI.

We now present the corresponding formal syntax.

The URI syntax specification [RFC3986] formally defines the IPv6 literal format in ABNF [RFC5234] by the following rule:

```
IP-literal = "[" ( IPv6address / IPvFuture  ) "]"
```

To provide support for a zone identifier, the existing syntax of IPv6address is retained, and a zone identifier may be added optionally to any literal address. This syntax allows flexibility for unknown future uses. The rule quoted above from [RFC3986] is replaced by three rules:

```
IP-literal = "[" ( IPv6address / IPv6addrz / IPvFuture  ) "]"
```

```
ZoneID = 1*( unreserved )
```

```
IPv6addrz = IPv6address "%" ZoneID
```

This change also applies to [RFC3987].

This syntax fills the gap that is described at the end of Section 11.7 of the IPv6 Scoped Address Architecture specification [RFC4007]. It replaces and obsoletes the syntax in Section 2 of [RFC6874].

The established rules for textual representation of IPv6 addresses [RFC5952] SHOULD be applied in producing URIs.

The URI syntax specification [RFC3986] states that URIs have a global scope, but that in some cases their interpretation depends on the end-user's context. URIs including a ZoneID are to be interpreted only in the context of the host at which they originate, since the ZoneID is of local significance only.

The IPv6 Scoped Address Architecture specification [RFC4007] offers guidance on how the ZoneID affects interface/address selection inside the IPv6 stack. Note that the behaviour of an IPv6 stack, if it is passed a non-null zone index for an address other than link-local, is undefined.

4. URI Parsers

This section discusses how URI parsers, such as those embedded in web browsers, might handle this syntax extension. Unfortunately, there is no formal distinction between the syntax allowed in a browser's input dialogue box and the syntax allowed in URIs. For this reason, no normative statements are made in this section.

In practice, although parsers respect the established syntax, they are coded pragmatically rather than being formally syntax-driven. Typically, IP address literals are handled by an explicit code path. Parsers have been inconsistent in providing for ZoneIDs. Most have no support, but there have been examples of ad hoc support. For example, some versions of Firefox allowed the use of a ZoneID preceded by a bare "%" character, but this feature was removed for consistency with established syntax [RFC3986]. As another example, some versions of Internet Explorer allowed use of a ZoneID preceded by a "%" character encoded as "%25", still beyond the syntax allowed by the established rules [RFC3986]. This syntax extension is in fact used internally in the Windows operating system and some of its APIs.

It is desirable for all URI parsers to recognise a ZoneID according to the syntax defined in Section 3.

URIs including a ZoneID have no meaning outside the originating HTTP client node. However, in some use cases, such as CUPS mentioned above, the URI will be reflected back to the client.

The various use cases for the ZoneID syntax will cause it to be entered in a browser's input dialogue box. Thus, URIs including a ZoneID are unlikely to occur in HTML documents. However, if they do (for example, in a diagnostic script coded in HTML), it would be appropriate to treat them exactly as above.

5. Security Considerations

The security considerations from the URI syntax specification [RFC3986] and the IPv6 Scoped Address Architecture specification [RFC4007] apply. In particular, this URI format creates a specific pathway by which a deceitful zone index might be communicated, as mentioned in the final security consideration of the Scoped Address Architecture specification.

To limit this risk, implementations MUST NOT allow use of this format except for well-defined usages, such as sending to link-local addresses under prefix fe80::/10. At the time of writing, this is the only well-defined usage known.

6. Acknowledgements

The lack of this format was first pointed out by Margaret Wasserman and later by Kerry Lynn. A previous draft document by Bill Fenner and Martin Dürst [LITERAL-ZONE] discussed this topic but was not finalised. Michael Sweet and Andrew Cady explained some of the difficulties caused by RFC 6874. The ABNF syntax proposed above was drafted by Andrew Cady.

Valuable comments and contributions were made by Karl Auer, Carsten Bormann, Benoit Claise, Martin Dürst, Stephen Farrell, Brian Haberman, Ted Hardie, Philip Homburg, Tatuya Jinmei, Yves Lafon, Barry Leiba, Radia Perlman, Tom Petch, Michael Richardson, Tomoyuki Sahara, Juergen Schoenwaelder, Nico Schottelius, Dave Thaler, Martin Thomson, Ole Troan, and others.

7. References

7.1. Normative References

- [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>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC3987] Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, DOI 10.17487/RFC3987, January 2005, <<https://www.rfc-editor.org/info/rfc3987>>.

- [RFC4007] Deering, S., Haberman, B., Jinmei, T., Nordmark, E., and B. Zill, "IPv6 Scoped Address Architecture", RFC 4007, DOI 10.17487/RFC4007, March 2005, <<https://www.rfc-editor.org/info/rfc4007>>.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<https://www.rfc-editor.org/info/rfc5234>>.
- [RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, DOI 10.17487/RFC5952, August 2010, <<https://www.rfc-editor.org/info/rfc5952>>.
- [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>>.

7.2. Informative References

- [CUPS] Apple, "CUPS open source printing system", 2021, <<https://www.cups.org/>>.
- [LITERAL-ZONE] Fenner, B. and M. Dürst, "Formats for IPv6 Scope Zone Identifiers in Literal Address Formats", Work in Progress, October 2005.
- [RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, DOI 10.17487/RFC3493, February 2003, <<https://www.rfc-editor.org/info/rfc3493>>.
- [RFC4001] Daniele, M., Haberman, B., Routhier, S., and J. Schoenwaelder, "Textual Conventions for Internet Network Addresses", RFC 4001, DOI 10.17487/RFC4001, February 2005, <<https://www.rfc-editor.org/info/rfc4001>>.
- [RFC6874] Carpenter, B., Cheshire, S., and R. Hinden, "Representing IPv6 Zone Identifiers in Address Literals and Uniform Resource Identifiers", RFC 6874, DOI 10.17487/RFC6874, February 2013, <<https://www.rfc-editor.org/info/rfc6874>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.

Appendix A. Options Considered

The syntax defined above allows a ZoneID to be added to any IPv6 address. The 6man WG discussed and rejected an alternative in which the existing syntax of IPv6address would be extended by an option to add the ZoneID only for the case of link-local addresses. It was felt that the solution presented in this document offers more flexibility for future uses and is more straightforward to implement.

The various syntax options considered are now briefly described.

1. Leave the problem unsolved.

This would mean that per-interface diagnostics would still have to be performed using ping or ping6:

```
ping fe80::abcd%en1
```

Advantage: works today.

Disadvantage: less convenient than using a browser. Leaves some use cases unsatisfied.

2. Simply use the percent character:

```
http://[fe80::abcd%en1]
```

Advantage: allows use of browser; allows cut and paste.

Disadvantage: requires code changes to all URI parsers.

This is the option chosen for standardisation.

3. Simply use an alternative separator:

```
http://[fe80::abcd-en1]
```

Advantage: allows use of browser; simple syntax.

Disadvantage: Requires all IPv6 address literal parsers and generators to be updated in order to allow simple cut and paste; inconsistent with existing tools and practice.

Note: The initial proposal for this choice was to use an underscore as the separator, but it was noted that this becomes effectively invisible when a user interface automatically underlines URLs.

4. Simply use the "IPvFuture" syntax left open in RFC 3986:

`http://[v6.fe80::abcd_en1]`

Advantage: allows use of browser.

Disadvantage: ugly and redundant; doesn't allow simple cut and paste.

5. Retain the percent character already specified for introducing zone identifiers for IPv6 Scoped Addresses [RFC4007], and then percent-encode it when it appears in a URI, according to the already-established URI syntax rules [RFC 3986]:

`http://[fe80::abcd%25en1]`

Advantage: allows use of browser; consistent with general URI syntax.

Disadvantage: somewhat ugly and confusing; doesn't allow simple cut and paste.

Appendix B. Change log

This section is to be removed before publishing as an RFC.

* draft-carpenter-6man-rfc6874bis-03, 2022-02-08:

- Changed to bare % signs.
- Added IRIs, RFC3987
- Editorial fixes

* draft-carpenter-6man-rfc6874bis-02, 2021-18-12:

- Give details of open issues
- Update authorship
- Editorial fixes

* draft-carpenter-6man-rfc6874bis-01, 2021-07-11:

- Added section on issues with RFC6874
- Removed suggested heuristic for bare % signs

- Editorial fixes
- * draft-carpenter-6man-rfc6874bis-00, 2021-07-05:
- Initial version

Authors' Addresses

Brian Carpenter
School of Computer Science
University of Auckland
PB 92019
Auckland 1142
New Zealand

Email: brian.e.carpenter@gmail.com

Stuart Cheshire
Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
United States of America

Email: cheshire@apple.com

Robert M. Hinden
Check Point Software
959 Skyway Road
San Carlos, CA 94070
United States of America

Email: bob.hinden@gmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 27 June 2022

B. Jordan, Ed.
Broadcom
S. Erdtman
Spotify AB
A. Rundgren
Independent
24 December 2021

JWS Clear Text JSON Signature Option (JWS/CT)
draft-jordan-jws-ct-07

Abstract

This document describes a method for extending the scope of the JSON Web Signature (JWS) specification, called JWS/CT (JWS "Clear Text"). By combining the detached mode of JWS with the JSON Canonicalization Scheme (JCS), JWS/CT enables JSON objects to remain in the JSON format after being signed. In addition to supporting a consistent data format, this arrangement also simplifies documentation, debugging, and logging. The ability to embed signed JSON objects in other JSON objects, makes the use of counter-signatures straightforward.

This informational specification has been produced outside the IETF, is not an IETF standard, and does not have IETF consensus. The intended audiences of this document are JSON tool vendors as well as designers of JSON-based cryptographic solutions.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 27 June 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction	3
2. Terminology	3
3. Detailed Operation	4
3.1. Signature Creation	4
3.1.1. Create the JSON Object to be Signed	4
3.1.2. Canonicalize the JSON Object to be Signed	5
3.1.3. Generate a JWS String	5
3.1.4. Assemble the Signed JSON Object	5
3.2. Signature Validation	6
3.2.1. Parse the Signed JSON Object	6
3.2.2. Fetch the Signature Property String	6
3.2.3. Remove the Signature Property String	6
3.2.4. Canonicalize the Remaining JSON Object	7
3.2.5. Validate the JWS String	7
4. IANA Considerations	8
5. Security Considerations	8
6. References	9
6.1. Normative References	9
6.2. Informative References	9
Appendix A. Open-Source Implementations	10
Appendix B. JWS/CT Application Notes	10
B.1. Counter-Signatures	10
B.2. Detached Signatures	12
B.3. Array of Signatures	13
Appendix C. Test Vector Using the ES256 Algorithm	14
Appendix D. Enhanced JWS Processing Option	15
Acknowledgements	15
Document History	15
Authors' Addresses	16

1. Introduction

This specification introduces a method for augmenting data expressed in the JSON [RFC8259] notation, with enveloped signatures, similar to the scheme used in XML Signature [XMLDSIG]. For interoperability reasons this specification constrains JSON objects to the I-JSON [RFC7493] subset.

To avoid "reinventing the wheel", this specification leverages JSON Web Signature (JWS) [RFC7515].

By building on the detached mode of JWS in combination with the JSON Canonicalization Scheme (JCS) [RFC8785], JSON objects to be signed can be kept in the JSON format. This arrangement is here referred to as JWS/CT, where CT stands for "Clear Text" signing.

The primary motivations for keeping signed JSON objects in the JSON format include simplified documentation, debugging, and logging, as well as for maintaining a consistent message structure.

Another target is HTTP-based signature schemes that currently utilize HTTP header values for holding detached signatures. By using the method described herein, signed JSON-formatted HTTP requests and responses may be self-contained and thus be serializable. The latter facilitates such data to be

- * stored in databases
- * passed through intermediaries
- * embedded in other JSON objects
- * counter-signed

without losing the ability to (at any time) verify signatures.

Appendix B outlines different ways to handle multiple signatures including counter-signing using JWS/CT.

The intended audiences of this document are JSON tool vendors as well as designers of JSON-based cryptographic solutions.

2. Terminology

Note that this document is not on the IETF standards track. However, a conformant implementation is supposed to adhere to the specified behavior for security and interoperability reasons. This text uses BCP 14 to describe that necessary behavior.

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.

3. Detailed Operation

This section describes the details related to signing and validating signatures based on this specification.

The following characteristics are crucial to know for prospective JWS/CT implementers and users:

- * With the exception of the reliance on the detached mode described in Appendix F of JWS [RFC7515], JWS/CT does not alter the JWS signature creation process, validation process, or format. This means that the contents of JWS headers as well as things related to signature algorithms and cryptographic keys are out of scope for this specification. A slightly enhanced processing option is outlined in Appendix D.
- * JWS/CT depends exclusively on the JWS Compact Serialization mode.
- * JSON data to be signed MUST be supplied as JSON objects. That is, direct signing of JSON arrays or JSON primitives is out of scope for this specification.
- * JCS [RFC8785] constrains JSON objects to the I-JSON [RFC7493] subset.

The signature creation and signature validation sections (Section 3.1 and Section 3.2 respectively), feature examples using the HS256 JOSE algorithm [RFC7518] with a 256-bit key having the following value, here expressed as hexadecimal bytes:

```
7f dd 85 1a 3b 9d 2d af c5 f0 d0 00 30 e2 2b 93
43 90 0c d4 2e de 49 48 56 8a 4a 2e e6 55 29 1a
```

3.1. Signature Creation

The following sub-sections describe how JSON objects can be signed according to the JWS/CT specification.

3.1.1. Create the JSON Object to be Signed

Create or parse the JSON object to be signed.

The following example object is used to illustrate the operations in the sections that follow:

```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true]
}
```

3.1.2. Canonicalize the JSON Object to be Signed

Use the result of the previous step as input to the canonicalization process described in JCS [RFC8785].

Applied to the example, the following JSON string should be generated:

```
{"otherProperties":[2000,true],"statement":"Hello signed world!"}
```

After encoding the string above in the UTF-8 [UNICODE] format, the following bytes (here in hexadecimal notation) should be generated:

```
7b 22 6f 74 68 65 72 50 72 6f 70 65 72 74 69 65 73 22 3a 5b 32 30
30 30 2c 74 72 75 65 5d 2c 22 73 74 61 74 65 6d 65 6e 74 22 3a 22
48 65 6c 6c 6f 20 73 69 67 6e 65 64 20 77 6f 72 6c 64 21 22 7d
```

3.1.3. Generate a JWS String

Use the result of the previous step as JWS Payload to the signature process described in Appendix F of JWS [RFC7515].

For the example, the JWS header is assumed to be:

```
{"alg":"HS256"}
```

The resulting JWS string should then after payload removal and using the key specified in Section 3, read as follows:

```
eyJhbGciOiJIUzI1NiJ9..VHVItCBcb8Q5CI-49imarDtJeSxH2uLU0DhqQP5Zjw4
```

3.1.4. Assemble the Signed JSON Object

Before a complete signed object can be created, a dedicated top-level property for holding the JWS signature string needs to be defined. The only requirement is that this property MUST NOT clash with any other top-level property name. The JWS string itself MUST be supplied as a JSON string argument to the signature property.

For the example, the property name "signature" is assumed to be the designated holder of the JWS string. Equipped with a signature property, the JWS string from the previous section, and the original JSON example, the process above should result in the following, now signed JSON object (with a line break in the "signature" property for display purposes only):

```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signature": "eyJhbGciOiJIUzI1NiJ9..VHVItCBCb8Q5CI-49imar
DtJeSxH2uLU0DhqQP5Zjw4"
}
```

3.2. Signature Validation

The following sub-sections describe how JSON objects signed according to the JWS/CT specification can be validated.

3.2.1. Parse the Signed JSON Object

Parse the JSON object that is expected to have been signed. If the parsing is unsuccessful, the operation MUST cause a compliant implementation to terminate processing and return an error indication.

To illustrate the subsequent operations the signed JSON object featured in Section 3.1.4 is used as example.

3.2.2. Fetch the Signature Property String

After successful parsing, retrieve the designated JSON top-level property holding the JWS string. If the property is missing or its argument is not a JSON string value, the operation MUST cause a compliant implementation to terminate processing and return an error indication.

For the example, where the property named "signature" is assumed to hold the JWS string, the operation above should return the following string:

```
eyJhbGciOiJIUzI1NiJ9..VHVItCBCb8Q5CI-49imarDtJeSxH2uLU0DhqQP5Zjw4
```

3.2.3. Remove the Signature Property String

Since the signature is calculated over the actual JSON object data, the designated signature property and its argument MUST be removed from the signed JSON object.

If applied to the example the resulting JSON object should read as follows:

```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true]
}
```

Note: JSON tools usually by default remove whitespace. In addition, the original ordering of properties may not always be honored. However, none of this has (due to the canonicalization performed by JCS), any impact on the result.

3.2.4. Canonicalize the Remaining JSON Object

Use the result of the previous step as input to the canonicalization process described in JCS [RFC8785].

If applied to the example the result of the process above should read as follows:

```
{"otherProperties":[2000,true],"statement":"Hello signed world!"}
```

After encoding the string above in the UTF-8 [UNICODE] format, the following bytes (here in hexadecimal notation) should be generated:

```
7b 22 6f 74 68 65 72 50 72 6f 70 65 72 74 69 65 73 22 3a 5b 32 30
30 30 2c 74 72 75 65 5d 2c 22 73 74 61 74 65 6d 65 6e 74 22 3a 22
48 65 6c 6c 6f 20 73 69 67 6e 65 64 20 77 6f 72 6c 64 21 22 7d
```

3.2.5. Validate the JWS String

After extracting the detached mode JWS string and canonicalizing the JSON object (to retrieve the JWS Payload), the JWS string MUST be restored as described in Appendix F of JWS [RFC7515]. The actual JWS validation procedure is not specified here because it is covered by [RFC7515] and also depends on application-specific policies like:

- * Accepted JWS signature algorithms
- * Accepted and/or required JWS header elements
- * Signature key lookup methods

If the validation process for some reason fails, the operation MUST cause a compliant implementation to terminate processing and return an error indication.

For the example, validation is straightforward since both the algorithm and the key to use are predefined (see Section 3). The input string to a JWS validator should after the process step above read as follows (with line breaks for display purposes only):

```
eyJhbGciOiJIUzI1NiJ9.eyJvdGhlclByb3BlcnRpZXMlOlsyMDAwLHRydWVdLCJzdGF0ZWllbnQiOiJIZWxsbyBzaWduZWQgd29ybGQhIn0.VHVItCBCb8Q5CI-49imarDtJeSxH2uLU0DhqQP5Zjw4
```

4. IANA Considerations

This document has no IANA actions.

5. Security Considerations

This specification inherits all the security considerations of JWS [RFC7515] and JCS [RFC8785].

In similarity to any other signature specification, it is crucial that signatures are verified before acting on the signed payload.

However, poorly tested software components may also introduce security issues. Consider the following JSON example:

```
{
  "fromAccount": "1234",
  "toAccount": "4567",
  "amount": {
    "value": 100,
    "currency": "USD"
  }
}
```

A non-compliant JCS implementation could return

```
{"amount": {}, "fromAccount": "1234", "toAccount": "4567"}
```

giving an attacker the ability to change "amount" to whatever it wants. Note though that this attack presumes that the consumer and producer use implementations broken in the same way, otherwise the signature would not validate.

For usage in a wider community, the name of the designated signature property becomes a critical factor that **MUST** be documented and communicated. However, in a properly designed system, a faulty or missing signature **MUST** "only" lead to failed operation, and not to a security breach.

6. References

6.1. Normative References

- [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>>.
- [RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <<https://www.rfc-editor.org/info/rfc7493>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.
- [RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<https://www.rfc-editor.org/info/rfc7518>>.
- [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>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <<https://www.rfc-editor.org/info/rfc8785>>.
- [UNICODE] The Unicode Consortium, "The Unicode Standard", <<https://www.unicode.org/versions/latest/>>.

6.2. Informative References

- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<https://www.rfc-editor.org/info/rfc7517>>.

- [RFC7797] Jones, M., "JSON Web Signature (JWS) Unencoded Payload Option", RFC 7797, DOI 10.17487/RFC7797, February 2016, <<https://www.rfc-editor.org/info/rfc7797>>.
- [SHS] NIST, "Secure Hash Standard (SHS)", FIPS PUB 180-4, August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>>.
- [XMLDSIG] W3C, "XML Signature Syntax and Processing Version 1.1", W3C Recommendation, April 2013, <<https://www.w3.org/TR/xmlsig-core1/>>.

Appendix A. Open-Source Implementations

Due to the simplicity of this specification, there is hardly a need for specific support software. However, JCS which is (at the time of writing), a relatively new design, may be fetched as a separate component for multiple platforms. The following open-source implementations have been verified to be compatible with JCS:

- * JavaScript: <<https://www.npmjs.com/package/canonicalize>>
- * Java: <<https://mvnrepository.com/artifact/io.github.erdtman/java-json-canonicalization>>
- * Go: <<https://github.com/cyberphone/json-canonicalization/tree/master/go>>
- * .NET/C#: <<https://github.com/cyberphone/json-canonicalization/tree/master/dotnet>>
- * Python: <<https://github.com/cyberphone/json-canonicalization/tree/master/python3>>

Appendix B. JWS/CT Application Notes

The following application notes are not a part of the JWS/CT core; they show how JWS/CT can be used in contexts involving multiple signatures.

B.1. Counter-Signatures

Consider the following JWS/CT object showing an imaginary real estate business record (with a line break in the "signature" property for display purposes only):

```
{
  "gps": [38.89768255588178, -77.03658644893932],
  "object": {
    "type": "house",
    "price": "$635,000"
  },
  "role": "buyer",
  "name": "John Smith",
  "timeStamp": "2020-11-08T13:56:08Z",
  "signature": "eyJhbGciOiJIUzI1NiJ9..z1PMniQiz4Eie86oK4xo25z
uyW92csiDqyiQrF6R5ug"
}
```

The signature above was created using the example key from Section 3.

Adding a notary signature on top of this could be performed by embedding the former object as follows (with line breaks in the "signature" properties for display purposes only):

```
{
  "attesting": {
    "gps": [38.89768255588178, -77.03658644893932],
    "object": {
      "type": "house",
      "price": "$635,000"
    },
    "role": "buyer",
    "name": "John Smith",
    "timeStamp": "2020-11-08T13:56:08Z",
    "signature": "eyJhbGciOiJIUzI1NiJ9..z1PMniQiz4Eie86oK4xo25z
uyW92csiDqyiQrF6R5ug"
  },
  "role": "notary",
  "name": "Carol Lombardi-Jones",
  "timeStamp": "2020-11-08T13:58:42Z",
  "signature": "eyJhbGciOiJFUzI1NiJ9..AVmJGUWp1JD0pf2jl_UQWXbf-
qj-2RWxOnyAXihd4POKbnjWqqSBmHPNfgMQFH_s5sXHkIOkDZe2nShqEJOEVA"
}
```

A side effect of this arrangement is that the notary's signature signs not only the notary data, but the buyer's data and signature as well. In most cases this way of adding signatures is advantageous since it maintains the actual order of signing events which also cannot be tampered with without invalidating the outermost signature.

Note that all properties above including "signature" are application specific.

The notary's signature was created using the example key from Appendix C.

B.2. Detached Signatures

In the case the signing entities are "peers" or are unrelated to each other, counter-signatures like described in Appendix B.1 are not applicable since they presume a specific flow. For supporting independent or asynchronous signers targeting a common document or data object, an imaginable solution is using a scheme where each signer calculates a hash of the target document/data and includes the hash together with signer-specific meta data like the following:

```
{
  <<Common Document/Data to Sign...>>

  "signers": [{
    "sha256": "<<Hash of Document/Data to Sign>>",

    <<Signer-related meta data...>>

    "signature": "<<Signer JWS Signature>>"
  }, {
    "sha256": "<<Hash of Document/Data to Sign>>",

    <<Signer-related meta data...>>

    "signature": "<<Signer JWS Signature>>"
  }]
}
```

In this case the object to sign would not be limited to JSON; it could, for example, be a PDF document hosted on a specific URL. Note that the relying party would have to update the structure for each signature received. In some cases a database would probably be more useful for holding individual signatures since a database can cope with any number of signers as well as keeping track of who have actually signed. The latter is crucial for things like international treaties and company board statements.

Note that although "signers", "sha256", and "signature" are application specific property names, the objects in the "signers" array are assumed to be fully conformant with the JWS/CT specification.

The following example shows a possible detached signature solution (with line breaks in the "signature" properties for display purposes only):

```
{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signers": [{
    "sha256": "n-i0HIBJKELoTicCK9c5nqJ8cYH0znGRcEbYKoQfm70",
    "timeStamp": "2020-11-18T07:45:28Z",
    "name": "Alice",
    "signature": "eyJhbGciOiJIUzI1NiJ9..AE7CnzSYsasPE3yrdsAwi
avd3IdWtdAmDE8FRMwYLA8"
  }, {
    "sha256": "n-i0HIBJKELoTicCK9c5nqJ8cYH0znGRcEbYKoQfm70",
    "timeStamp": "2020-11-18T08:03:40Z",
    "name": "Bob",
    "signature": "eyJhbGciOiJIUzI1NiJ9..0tNLy0pLcHUjPhhorpKd5
7a8zTPeqlrOjATiSlPQlvcIE99x6mHmow04tPbJS8dqSqO9c4RkKW6jeL4ZyWpXLA"
  }]
}
```

Notes:

- * "Alice" used the example key from Section 3 while "Bob" used the example key specified in Appendix C.
- * The "sha256" properties hold base64url-encoded [RFC4648], SHA256-hashes [SHS] of the canonicalized data created in Section 3.1.2.
- * This arrangement requires a two-step validation process where each JWS/CT object in the "signers" array is individually validated, as well as having its "sha256" property compared with the actual hash of the canonicalized common data.

B.3. Array of Signatures

Another possibility supporting multiple and independent signatures is collecting JWS signature strings in a JSON array object according to the following scheme:


```

{
  <<Common Document/Data to Sign...>>

  "<<Signature property>>": [ "<<Signature-1>>",
                                "<<Signature-2>>",
                                .
                                "<<Signature-n>>" ]
}

```

Processing would follow Section 3, with the addition that each signature is dealt with individually.

Compared to Appendix B.2, signature arrays imply that possible signer-specific meta-data is supplied as JWS extensions in the associated signature's base64url-encoded header.

By combining the example used in Section 3 with the test vector in Appendix C, a valid signature array object could be as follows (with line breaks in the "signatures" property for display purposes only):

```

{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signatures": ["eyJhbGciOiJIUzI1NiJ9..VHVItCBCb8Q5CI-49imar
DtJeSxH2uLU0DhqQP5Zjw4",
                 "eyJhbGciOiJFUzI1NiJ9..ENP0j0-QPsA7N_Mg1-RMN
9IxapeTWtQwR7sPUqEiSNHPuV_fqSdRqqkLOlBdV0lcc4lSJdn1XCv-ZHYdZ9t3kA"]
}

```

Note that "signatures" is not a keyword, it was only selected to highlight the fact that there are multiple signatures.

Appendix C. Test Vector Using the ES256 Algorithm

This appendix shows how a signed version of the JSON example object in Section 3.1.1 would look like if applying the ES256 JOSE algorithm [RFC7518] (with a line break in the "signature" property for display purposes only):

```

{
  "statement": "Hello signed world!",
  "otherProperties": [2000, true],
  "signature": "eyJhbGciOiJFUzI1NiJ9..ENP0j0-QPsA7N_Mg1-RMN
9IxapeTWtQwR7sPUqEiSNHPuV_fqSdRqqkLOlBdV0lcc4lSJdn1XCv-ZHYdZ9t3kA"
}

```

The example above depends on a JWS header holding the algorithm {"alg":"ES256"}, and the following private key, here expressed in the JWK [RFC7517] format:

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "6BKxpty8cI-exDzCkh-goU6dXq3MbcY0cd1LaAxiNrU",
  "y": "mChcvUzm44j3Lt2b5BPyQloQ91tf2D2V-gzeUxWaUdg",
  "d": "6XxMFXhcYT5QN9w5TIg2aSKsbcj-pj4BnZkK7Zot4B8"
}
```

Note that signing with the ES256 algorithm returns different results for each signature due to a randomization step in the signature computation process.

Appendix D. Enhanced JWS Processing Option

By default, JWS/CT uses the JWS compact serialization mode "as is". As a consequence, a technically redundant, internal-only, base64url encoding step is performed over the JWS Payload. Although the performance hit should be marginal for most real-world applications, a possibility is using the "Unencoded Payload" mode of RFC7797 [RFC7797]. However, this requires that the JWS implementation supports the "b64":false and "crit":["b64"] header elements implied by RFC7797, effectively rendering the RFC7797 mode as an implementer option for specific communities.

Acknowledgements

People who have contributed directly and indirectly with valuable input to this specification include Vladimir Dzhuvinov, Freddi Gyara, and Filip Skokan.

Document History

[[This section to be removed by the RFC Editor before publication as an RFC]]

Version 00:

- * Initial publication.

Version 01:

- * Added paragraph to Abstract.

- * Updated Security Considerations.

Version 02:

- * Changed alternative test key to ES256/P-256.
- * Moved RFC7797 to an appendix.
- * Changed <tt> to only be used on keywords.
- * Added some clarity to detached signatures.

Version 03:

- * Language changes suggested by ISE.

Version 04:

- * Language nit.

Version 05:

- * Document refresh.

Version 06:

- * Changes after ISE review.

Version 07:

- * Changes after ISE and external reviews.

Authors' Addresses

Bret Jordan (editor)
Broadcom
1320 Ridder Park Drive
San Jose, CA 95131
United States of America

Email: bret.jordan@broadcom.com

Samuel Erdtman
Spotify AB
Birger Jarlsgatan 61, 4tr
SE-113 56 Stockholm
Sweden

Email: erdtman@spotify.com

Anders Rundgren
Independent
Montpellier
France

Email: anders.rundgren.net@gmail.com

URI: <https://www.linkedin.com/in/andersrundgren/>

TODO Working Group
Internet-Draft
Intended status: Informational
Expires: 8 September 2022

K. Pugin
A. Frindell
J. Cenzano
J. Weissman
Facebook
7 March 2022

RUSH - Reliable (unreliable) streaming protocol
draft-kpugin-rush-01

Abstract

RUSH is an application-level protocol for ingesting live video. This document describes the protocol and how it maps onto QUIC.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the mailing list (), which is archived at .

Source for this draft and an issue tracker can be found at <https://github.com/afrind/draft-rush>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
2. Conventions and Definitions	3
3. Theory of Operations	3
3.1. Connection establishment	3
3.2. Sending Video Data	4
3.3. Receiving data	4
3.4. Reconnect	5
4. Wire Format	5
4.1. Frame Header	5
4.2. Frames	8
4.2.1. Connect frame	8
4.2.2. Connect Ack frame	8
4.2.3. End of Video frame	9
4.2.4. Error frame	9
4.2.5. Video frame	10
4.2.6. Audio frame	11
4.2.7. GOAWAY frame	13
4.3. QUIC Mapping	13
4.3.1. Normal mode	13
4.3.2. Multi Stream Mode	13
5. Error Handling	14
5.1. Connection Errors	14
5.2. Frame errors	14
6. Extensions	15
7. Security Considerations	15
8. IANA Considerations	15
9. Normative References	15
Acknowledgments	16
Authors' Addresses	16

1. Introduction

RUSH is a bidirectional application level protocol designed for live video ingestion that runs on top of QUIC.

RUSH was built as a replacement for RTMP (Real-Time Messaging Protocol) with the goal to provide support for new audio and video codecs, extensibility in the form of new message types, and multi-track support. In addition, RUSH gives applications option to control data delivery guarantees by utilizing QUIC streams.

This document describes the RUSH protocol, wire format, and QUIC mapping.

2. Conventions and Definitions

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.

Frame/Message: logical unit of information that client and server can exchange

PTS: presentation timestamp

DTS: decoding timestamp

AAC: advanced audio codec

NALU: network abstract layer unit

VPS: video parameter set (H265 video specific NALU)

SPS: sequence parameter set (H264/H265 video specific NALU)

PPS: picture parameter set (H264/H265 video specific NALU)

ADTS header: _Audio Data Transport Stream Header_

ASC: Audio specific config

GOP: Group of pictures, specifies the order in which intra- and inter-frames are arranged.

3. Theory of Operations

3.1. Connection establishment

In order to live stream using RUSH, the client establishes a QUIC connection using the ALPN token "rush".

After the QUIC connection is established, client creates a new bidirectional QUIC stream, chooses starting frame ID and sends Connect frame Section 4.2.1 over that stream. This stream is called the Connect Stream.

The client sends mode of operation setting in Connect frame payload, format of the payload is TBD.

One connection SHOULD only be used to send one video.

3.2. Sending Video Data

The client can choose to wait for the ConnectAck frame Section 4.2.2 or it can start sending data immediately after sending the Connect frame.

A track is a logical organization of the data, for example, video can have one video track, and two audio tracks (for two languages). The client can send data for multiple tracks simultaneously.

The encoded audio or video data of each track is serialized into frames (see Section 4.2.6 or Section 4.2.5) and transmitted from the client to the server. Each track has its own monotonically increasing frame ID sequence. The client MUST start with initial frame ID = 1.

Depending on mode of operation (Section 4.3), the client sends audio and video frames on the Connect stream or on a new QUIC stream for each frame.

In Multi Stream Mode (Section 4.3.2), the client can stop sending a frame by resetting the corresponding QUIC stream. In this case, there is no guarantee that the frame was received by the server.

3.3. Receiving data

Upon receiving Connect frame, the server replies with ConnectAck frame Section 4.2.2 and prepares to receive audio/video data.

It's possible that in Multi Stream Mode (Section 4.3.2), the server receives audio or video data before it receives the Connect frame. The implementation can choose whether to buffer or drop the data. The audio/video data cannot be interpreted correctly before the arrival of the Connect frame.

In Normal Mode (Section 4.3.1), it is guaranteed by the transport that frames arrive into the application layer in order they were sent.

In Multi Stream Mode, it's possible that frames arrive at the application layer in a different order than they were sent, therefore the server MUST keep track of last received frame ID for every track that it receives. A gap in the frame sequence ID on a given track can indicate out of order delivery and the server MAY wait until missing frames arrive. The server must consider frame lost if the corresponding QUIC stream was reset.

Upon detecting a gap in the frame sequence, the server MAY wait for the missing frames to arrive for an implementation defined time. If missing frames don't arrive, the server SHOULD consider them lost and continue processing rest of the frames. For example if the server receives the following frames for track 1: 1 2 3 5 6 and frame #4 hasn't arrived after implementation defined timeout, the server SHOULD continue processing frames 5 and 6.

When the client is done streaming, it sends the End of Video frame (Section 4.2.3) to indicate to the server that there won't be any more data sent.

3.4. Reconnect

If the QUIC connection is closed at any point, client MAY reconnect by simply repeat the Connection establishment process (Section 3.1) and resume sending the same video where it left off. In order to support termination of the new connection by a different server, the client SHOULD resume sending video frames starting with I-frame, to guarantee that the video track can be decoded.

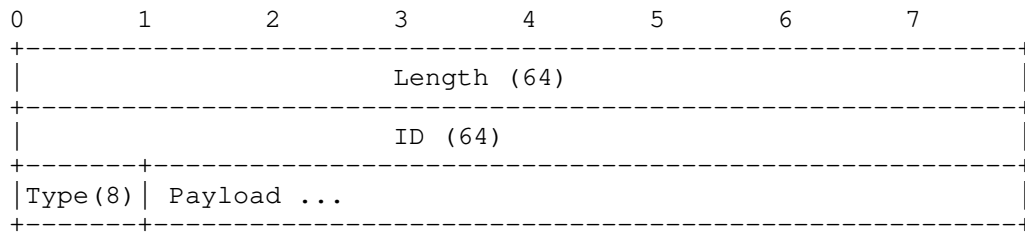
Reconnect can be initiated by the server if it needs to "go away" for maintenance. In this case, the server sends a GOAWAY frame (Section 4.2.7) to advise the client to gracefully close the connection. This allows client to finish sending some data and establish new connection to continue sending without interruption.

4. Wire Format

4.1. Frame Header

The client and server exchange information using frames. There are different types of frames and the payload of each frame depends on its type.

Generic frame format:



Length(64): Each frame starts with length field, 64 bit size that tells size of the frame in bytes (including predefined fields, so if LENGTH is 100 bytes, then PAYLOAD length is $100 - 8 - 8 - 1 = 82$ bytes).

ID(64): 64 bit frame sequence number, every new frame MUST have a sequence ID greater than that of the previous frame within the same track. Track ID would be specified in each frame. If track ID is not specified it's 0 implicitly.

Type(8): 1 byte representing the type of the frame.

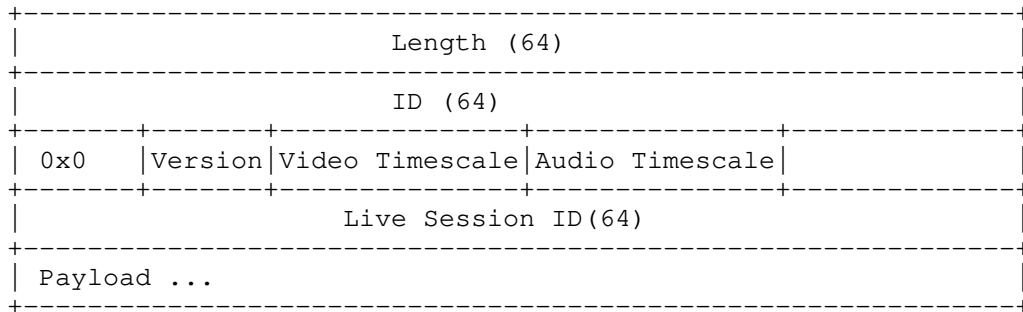
Predefined frame types:

Frame Type	Frame
0x0	connect frame
0x1	connect ack frame
0x2	reserved
0x3	reserved
0x4	end of video frame
0x5	error frame
0x6	reserved
0x7	reserved
0x8	reserved
0x9	reserved
0xA	reserved
0xB	reserved
0xC	reserved
0xD	video frame
0xE	audio frame
0xF	reserved
0x10	reserved
0x11	reserved
0x12	reserved
0x13	reserved
0x14	GOAWAY frame

Table 1

4.2. Frames

4.2.1. Connect frame



Version: version of the protocol (initial version is 0x0).

Video Timescale: timescale for all video frame timestamps on this connection. Recommended value 30000

Audio Timescale: timescale for all audio samples timestamps on this connection, recommended value same as audio sample rate, for example 44100

Live Session ID: identifier of broadcast, when reconnect, client MUST use the same live session ID

Payload: application and version specific data that can be used by the server. OPTIONAL

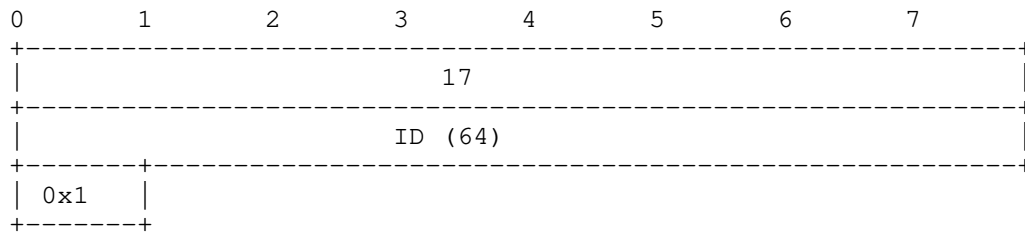
This frame is used by the client to initiate broadcasting. The client can start sending other frames immediately after "Connect frame" without waiting acknowledgement from the server.

If server doesn't support VERSION sent by the client, the server sends an Error frame with code UNSUPPORTED VERSION.

If audio timescale or video timescale are 0, the server sends error frame with error code INVALID FRAME FORMAT and closes connection.

If the client receives a Connect frame from the server, the client sends an Error frame with code TBD.

4.2.2. Connect Ack frame



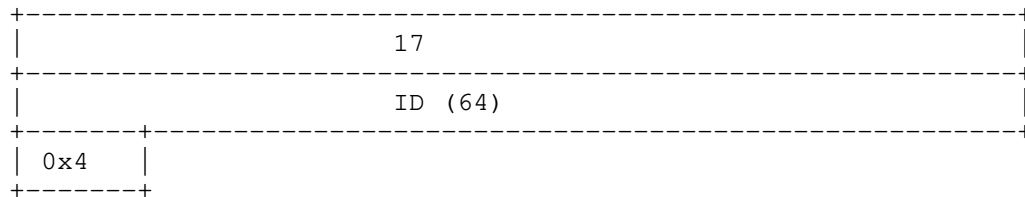
The server sends the "Connect Ack" frame in response to "Connect" frame indicating that server accepts "version" and is ready to receive data.

If the client doesn't receive "Connect Ack" frame from the server within a timeout, it will close the connection. The timeout value is chosen by the implementation.

There can be only one "Connect Ack" frame sent over lifetime of the QUIC connection.

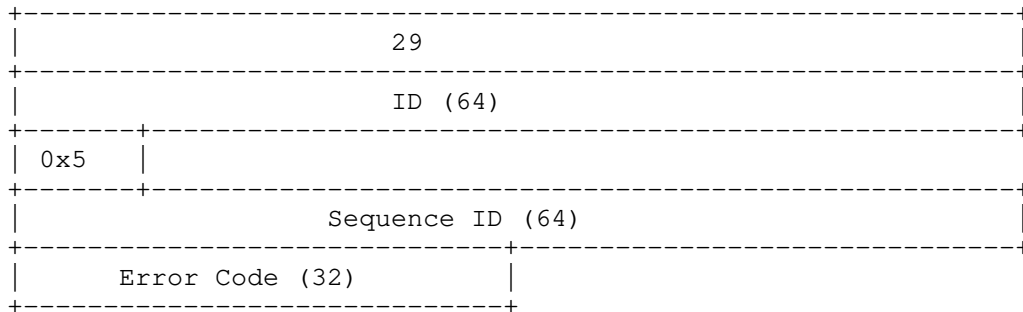
If the server receives a Connect Ack frame from the client, the client sends an Error frame with code TBD.

4.2.3. End of Video frame



End of Video frame is sent by a client when it's done sending data and is about to close the connection. The server SHOULD ignore all frames sent after that.

4.2.4. Error frame



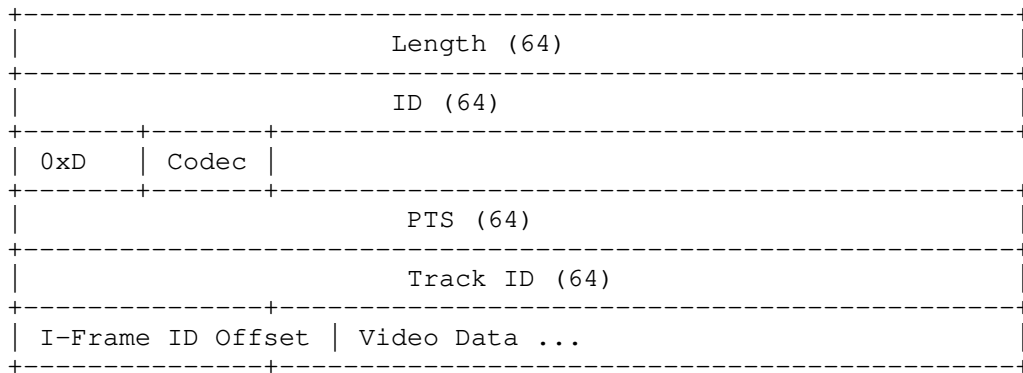
Sequence ID: ID of the frame sent by the client that error is generated for, ID=0x0 indicates connection level error.

Error Code: 32 bit unsigned integer

Error frame can be sent by the client or the server to indicate that an error occurred.

Some errors are fatal and the connection will be closed after sending the Error frame.

4.2.5. Video frame



Codec: specifies codec that was used to encode this frame.

PTS: presentation timestamp in connection video timescale

DTS: decoding timestamp in connection video timescale

Supported type of codecs:

Type	Codec
0x1	H264
0x2	H265
0x3	VP8
0x4	VP9

Table 2

Track ID: ID of the track that this frame is on

I-Frame ID Offset: Distance from sequence ID of the I-frame that is required before this frame can be decoded. This can be useful to decide if frame can be dropped.

Video Data: variable length field, that carries actual video frame data that is codec dependent

For h264/h265 codec, "Video Data" are 1 or more NALUs in AVCC format:

0	1	2	3	4	5	6	7
NALU Length (64)							
NALU Data ...							

EVERY h264 video key-frame MUST start with SPS/PPS NALUs. EVERY h265 video key-frame MUST start with VPS/SPS/PPS NALUs.

Binary concatenation of "video data" from consecutive video frames, without data loss MUST produce VALID h264/h265 bitstream.

4.2.6. Audio frame

Length (64)	
ID (64)	
0xE	Codec
Timestamp (64)	
TrackID	
Audio Data ...	

Codec: specifies codec that was used to encode this frame.

Supported type of codecs:

Type	Codec
0x1	AAC
0x2	OPUS

Table 3

Timestamp: timestamp of first audio sample in Audio Data.

Track ID: ID of the track that this frame is on

Audio Data: variable length field, that carries 1 or more audio frames that is codec dependent.

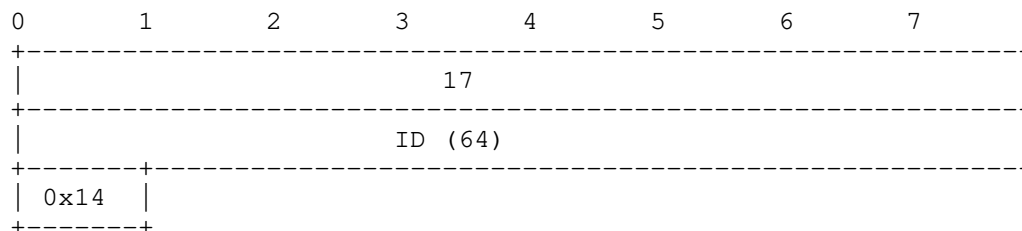
For AAC codec, "Audio Data" are 1 or more AAC samples, prefixed with ADTS HEADER:

152	158	...	N
ADTS(56)	AAC SAMPLE		

Binary concatenation of all AAC samples in "Audio Data" from consecutive audio frames, without data loss MUST produce VALID AAC bitstream.

For OPUS codec, "Audio Data" are 1 or more OPUS samples, prefixed with OPUS header as defined in [RFC7845]

4.2.7. GOAWAY frame



The GOAWAY frame is used by the server to initiate graceful shutdown of a connection, for example, for server maintenance.

Upon receiving GOAWAY, the client MUST send frames remaining in current GOP and stop sending new frames on this connection. The client SHOULD establish a new connection and resume sending frames there.

After sending a GOAWAY frame, the server continues processing arriving frames for an implementation defined time, after which the server SHOULD close the connection.

4.3. QUIC Mapping

One of the main goals of the RUSH protocol was ability to provide applications a way to control reliability of delivering audio/video data. This is achieved by using a special mode Section 4.3.2.

4.3.1. Normal mode

In normal mode, RUSH uses one bidirectional QUIC stream to send data and receive data. Using one stream guarantees reliable, in-order delivery - applications can rely on QUIC transport layer to retransmit lost packets. The performance characteristics of this mode are similar to RTMP over TCP.

4.3.2. Multi Stream Mode

In normal mode, if packet belonging to video frame is lost, all packets sent after it will not be delivered to application, even though those packets may have arrived at the server. This introduces head of line blocking and can negatively impact latency.

To address this problem, RUSH defines "Multi Stream Mode", in which one QUIC stream is used per audio/video frame.

Connection establishment follows the normal procedure by client sending Connect frame, after that Video and Audio frames are sent using following rules:

- * Each new frame is sent on new bidirectional QUIC stream
- * Frames within same track must have IDs that are monotonically increasing, such that $ID(n) = ID(n-1) + 1$

The receiver reconstructs the track using the frames IDs.

Response Frames (Connect Ack and Error), will be in the response stream of the stream that sent it.

The client MAY control delivery reliability by setting a delivery timer for every audio or video frame and reset the QUIC stream when the timer fires. This will effectively stop retransmissions if the frame wasn't fully delivered in time.

Timeout is implementation defined, however future versions of the draft will define a way to negotiate it.

5. Error Handling

An endpoint that detects an error SHOULD signal the existence of that error to its peer. Errors can affect an entire connection (see Section 5.1), or a single frame (see Section 5.2).

The most appropriate error code SHOULD be included in the error frame that signals the error.

5.1. Connection Errors

There is one error code defined in core of the protocol that indicates connection error:

1 - UNSUPPORTED VERSION - indicates that the server doesn't support version specified in Connect frame

5.2. Frame errors

There are two error codes defined in core protocol that indicate a problem with a particular frame:

2 - UNSUPPORTED CODEC - indicates that the server doesn't support the given audio or video codec

3 - INVALID FRAME FORMAT - indicates that the receiver was not able to parse the frame or there was an issue with a field's value.

6. Extensions

RUSH permits extension of the protocol.

Extensions are permitted to use new frame types (Section 4), new error codes (Section 4.2.4), or new audio and video codecs (Section 4.2.6, Section 4.2.5).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements, except codec id, which returns an UNSUPPORTED CODEC error. Implementations MUST discard frames that have unknown or unsupported types.

7. Security Considerations

RUSH protocol relies on security guarantees provided by the transport.

Implementation SHOULD be prepare to handle cases when sender deliberately sends frames with gaps in sequence IDs.

Implementation SHOULD be prepare to handle cases when server never receives Connect frame (Section 4.2.1).

A frame parser MUST ensure that value of frame length field (see Section 4.1) matches actual length of the frame, including the frame header.

Implementation SHOULD be prepare to handle cases when sender sends a frame with large frame length field value.

8. IANA Considerations

TODO: add frame type registry, error code registry, audio/video codecs registry

9. Normative References

[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/rfc/rfc2119>>.

- [RFC7845] Terriberry, T., Lee, R., and R. Giles, "Ogg Encapsulation for the Opus Audio Codec", RFC 7845, DOI 10.17487/RFC7845, April 2016, <<https://www.rfc-editor.org/rfc/rfc7845>>.
- [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/rfc/rfc8174>>.

Acknowledgments

This draft is the work of many people: Vlad Shubin, Nitin Garg, Milen Lazarov, Benny Luo, Nick Ruff, Konstantin Tsoy, Nick Wu.

Authors' Addresses

Kirill Pugin
Facebook
Email: ikir@fb.com

Alan Frindell
Facebook
Email: afrind@fb.com

Jordi Cenzano
Facebook
Email: jccenzano@fb.com

Jake Weissman
Facebook
Email: jakeweissman@fb.com

Network Working Group
Internet-Draft
Obsoletes: 4568 (if approved)
Updates: 7201 (if approved)
Intended status: Standards Track
Expires: January 13, 2022

J. Preuss Mattsson
M. Westerlund
Ericsson
July 12, 2021

SDP Security Descriptions is NOT RECOMMENDED and Historic
draft-mattsson-dispatch-sdes-dont-dont-dont-00

Abstract

Key exchange without forward secrecy enables pervasive monitoring. Massive pervasive monitoring attacks relying on key exchange without forward secrecy have been reported, and many more have likely happened without ever being reported. If key exchange without Diffie-Hellman is used, access to long-term keys enable passive attackers to compromise past and future sessions. Entities can get access to long-term key material in different ways: physical attacks, hacking, social engineering attacks, espionage, or by simply demanding access to keying material with or without a court order. Session Description Protocol (SDP) Security Descriptions (RFC 4568) does not offer PFS and has a large number of additional significant security weaknesses. This document specifies that use of the SDP Security Descriptions is NOT RECOMMENDED. New deployments SHOULD forbid support of SDP Security Descriptions.

This document reclassifies RFC 4568 (SDP Security Descriptions) to Historic Status and also obsoletes RFC 4568.

This document updates RFC 7201 (Options for Securing RTP Sessions) to note that SDP Security Descriptions SHOULD NOT be used.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 13, 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Terminology	4
2. Status Change	4
3. References	5
3.1. Normative References	5
3.2. Informative References	6
Acknowledgements	8
Authors' Addresses	8

1. Introduction

Key exchange without forward secrecy enables pervasive monitoring [RFC7258]. Massive pervasive monitoring attacks relying on key exchange without forward secrecy have been reported [Heist] [I-D.ietf-emu-aka-pfs], and many more have likely happened without ever being reported. If key exchange without Diffie-Hellman is used, access to long-term keys enables passive attackers to compromise past and future sessions. Entities can get access to long-term key material in different ways: physical attacks, hacking, social engineering attacks, espionage, or by simply demanding access to keying material with or without a court order.

All TLS cipher suites without forward secrecy has been marked as NOT RECOMMENDED [RFC8447] in TLS 1.2 [RFC5246], and static RSA and DH are forbidden in TLS 1.3 [RFC8446]. A large number of TLS profiles forbid use of key exchange without Diffie-Hellman for TLS 1.2 [RFC7540], [ANSSI], [T3GPP.33.210]. SRTP deployments are severely lagging behind TLS deployments in this area as SDP Security Descriptions [RFC4568] is still used in many deployments. SDP

Security Description is often referred to as SDES. In this document SDES refers to SDP Security Descriptions and not RTP source descriptions, which are also often referred to as SDES.

In addition to the very serious weaknesses of not providing protection against key leakage and enabling passive monitoring [RFC7258], Security Descriptions [RFC4568] has a number of additional significant security problems.

- o As stated in [RFC7201], Security Descriptions is vulnerable to SSRC collisions, which leads to so called "two-time pad" attacks. This vulnerability is worse than using a 32-bit MAC for data integrity, as a "two-time pad" may lead to loss of both confidentiality and integrity.
- o In addition to happening by itself with a non-negligible probability, the SSRC collision attack can also be triggered by an attacker. As stated in [Replay-SDES] "The most serious attack is a replay attack on SDES, which causes SRTP to repeat the keystream used for media encryption, thus completely breaking transport-layer security." The authors stress that the attack is practical in existing implementations: "We emphasize that our attack on SDES/SRTP is not a theoretical exercise.". If SSRC are predictable, an attacker can also improve the probability by blocking SDP with non-colliding SSRCs.
- o Security Descriptions [RFC4568] requires use of an encapsulating data-security protocol on each hop in the path giving at best hop-by-hop security. This assumes that all the intermediaries are trusted and uncompromised. This is a very insecure and outdated security model that fails completely as soon as a single node in the path is compromised.
- o While Security Descriptions [RFC4568] requires use of an encapsulating data-security protocol, several deployed systems are known to use Security Descriptions without any encapsulating data-security protocol to protect the SDP messages. This means that any on-path attacker can decrypt the communication as well as modify and inject SRTP packets. A huge problem with SDP Security Descriptions is that the endpoints have no way of verifying if the path is protected or not.
- o As explained in [Baiting-SDES] the model of slitting the security between two independent layers is flawed, SDP Security Description is vulnerable to the Baiting attack [I-D.kaplan-sip-baiting-attack], and "This situation leads to security vulnerability and attacker could get master key by spoofing in unencrypted path."

- o As Security Descriptions [RFC4568] transports cryptographic keys without confidentiality in Session Description Protocol, the media keys are available to any entity that has visibility to the SDP [RFC5411]. The cryptographic keys are known to often end up in logs and data retention systems. These systems are often accessible by many more user accounts than Lawful Interception (LI) systems.
- o [Hacking-SDES] summarizes the security of SDP Security Descriptions as "the false sense of security might be more dangerous than simply leaving your voice calls unencrypted."

New systems and recommendations like WebRTC [RFC8827], PERC [RFC8871], and [RFC8862] do mandate support of DTLS-SRTP [RFC5764]. WebRTC also forbids support of SDP Security Descriptions.

- o WebRTC [RFC8827] states (for good reasons) that "WebRTC implementations MUST NOT offer SDP security descriptions [RFC4568] or select it if offered."

Many implementations, devices, and libraries support DTLS-SRTP. One deployment that supports DTLS-SRTP but still use SDP Security Descriptions is IMS Media Security [T3GPP.33.328] where Security Descriptions is one option used for access protection. IMS Media Security with SDP Security Descriptions is typically used for VoWi-Fi (Voice over EPC-integrated Wi-Fi) calls which is commonly used by 4G and 5G phones as a backup to VoLTE (Voice over LTE) and VoNR (Voice over NR). However, IMS Media Security [T3GPP.33.328] has already specified and mandated support of DTLS-SRTP for interworking with WebRTC. Allowing use of DTLS-SRTP also for other use cases than WebRTC interworking would therefore be a relatively small change.

1.1. 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.

2. Status Change

This document reclassifies RFC 4568 (SDP Security Descriptions) to Historic Status and also obsoletes RFC 4568.

This document updates RFC 7201 (Options for Securing RTP Sessions) to note that SDP Security Descriptions SHOULD NOT be used.

This document specifies that use of the SDP Security Descriptions [RFC4568] is NOT RECOMMENDED. Existing deployments SHOULD mandate support of DTLS-SRTP [RFC5764] and long-term phase out use of SDP Security Descriptions. If it is known by out-of-band means that the other party supports DTLS-SRTP, then SDP Security Descriptions MUST NOT be offered or accepted. If it is not known if the other party supports DTLS-SRTP, both DTLS-SRTP and SDP Security Descriptions and SHOULD be offered during a transition period. New deployments SHOULD forbid support of Security Descriptions [RFC4568]. This document reclassifies RFC 4568, SDP Security Descriptions to Historic Status and obsoletes RFC 4568.

As required by [RFC7258], work on IETF protocols needs to consider the effects of pervasive monitoring and mitigate them when possible.

3. References

3.1. Normative References

- [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>>.
- [RFC4568] Andreasen, F., Baugher, M., and D. Wing, "Session Description Protocol (SDP) Security Descriptions for Media Streams", RFC 4568, DOI 10.17487/RFC4568, July 2006, <<https://www.rfc-editor.org/info/rfc4568>>.
- [RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <<https://www.rfc-editor.org/info/rfc5764>>.
- [RFC7201] Westerlund, M. and C. Perkins, "Options for Securing RTP Sessions", RFC 7201, DOI 10.17487/RFC7201, April 2014, <<https://www.rfc-editor.org/info/rfc7201>>.
- [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>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

- [RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <<https://www.rfc-editor.org/info/rfc8447>>.
- [RFC8862] Peterson, J., Barnes, R., and R. Housley, "Best Practices for Securing RTP Media Signaled with SIP", BCP 228, RFC 8862, DOI 10.17487/RFC8862, January 2021, <<https://www.rfc-editor.org/info/rfc8862>>.

3.2. Informative References

- [ANSSI] Agence nationale de la securite des systemes d'information, ., "Security Recommendations for TLS", January 2017, <https://www.ssi.gouv.fr/uploads/2017/02/security-recommendations-for-tls_v1.1.pdf>.
- [Baiting-SDS]
Seokung Yoon, ., Jongil Jeong, ., and . Hyuncheol Jeong, "A Study on the Tightening the Security of the Key Management Protocol (RFC4568) for VoIP", June 2010, <<http://phbo.janlo.nl/kpngips2/module-09/security-analysis.pdf>>.
- [Hacking-SDS]
Anthony Critelli, ., "Hacking VoIP: Decrypting SDS Protected SRTP Phone Calls", June 2014, <<https://www.acritelli.com/blog/hacking-voip-decrypting-sdes-protected-srtp-phone-calls/>>.
- [Heist] The Intercept, ., "How Spies Stole the Keys to the Encryption Castle", February 2015, <<https://theintercept.com/2015/02/19/great-sim-heist/>>.
- [I-D.ietf-emu-aka-pfs]
Arkko, J., Norrman, K., and V. Torvinen, "Perfect-Forward Secrecy for the Extensible Authentication Protocol Method for Authentication and Key Agreement (EAP-AKA' PFS)", draft-ietf-emu-aka-pfs-05 (work in progress), October 2020.
- [I-D.kaplan-sip-baiting-attack]
Kaplan, H., Wing, D., and I. Property, "The SIP Identity Baiting Attack", draft-kaplan-sip-baiting-attack-02 (work in progress), February 2008.

[Replay-SDES]

Prateek Gupta, . and . Vitaly Shmatikov, "Security Analysis of Voice-over-IP Protocols", June 2007, <<http://phbo.janlo.nl/kpngips2/module-09/security-analysis.pdf>>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.

[RFC5411] Rosenberg, J., "A Hitchhiker's Guide to the Session Initiation Protocol (SIP)", RFC 5411, DOI 10.17487/RFC5411, February 2009, <<https://www.rfc-editor.org/info/rfc5411>>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <<https://www.rfc-editor.org/info/rfc7258>>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

[RFC8827] Rescorla, E., "WebRTC Security Architecture", RFC 8827, DOI 10.17487/RFC8827, January 2021, <<https://www.rfc-editor.org/info/rfc8827>>.

[RFC8871] Jones, P., Benham, D., and C. Groves, "A Solution Framework for Private Media in Privacy-Enhanced RTP Conferencing (PERC)", RFC 8871, DOI 10.17487/RFC8871, January 2021, <<https://www.rfc-editor.org/info/rfc8871>>.

[T3GPP.33.210]

3GPP, ., "TS 33.210 Network Domain Security (NDS); IP network layer security", July 2020, <<https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2279>>.

[T3GPP.33.328]

3GPP, ., "TS 33.328 IP Multimedia Subsystem (IMS) media plane security", April 2021, <<https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2295>>.

Acknowledgements

The authors want to thank Bo Burman and Christer Holmberg for their valuable comments and feedback.

Authors' Addresses

John Preuss Mattsson
Ericsson

Email: john.mattsson@ericsson.com

Magnus Westerlund
Ericsson

Email: magnus.westerlund@ericsson.com

dispatch
Internet-Draft
Intended status: Informational
Expires: 7 January 2022

J. Oreland
H. Alvestrand
Google
6 July 2021

NICER - a better usage profile on ICE
draft-oreland-dispatch-ice-nicer-00

Abstract

NICER presents an usage profile of ICE that permits more dynamic adaptation to network conditions over the time of a call.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the mailing list (dispatch@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/browse/dispatch/>.

Source for this draft and an issue tracker can be found at <https://github.com/alvestrand/nicer-spec>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 7 January 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Conventions and Definitions	2
3. Problem statement	2
4. Traditional ICE	3
5. NICER - the idea	3
6. Standardization requirements	4
7. Possible optimizations	4
8. Implementation issues	4
9. Security Considerations	5
10. IANA Considerations	5
11. Normative References	5
Acknowledgments	6
Authors' Addresses	6

1. Introduction

TODO Introduction

2. Conventions and Definitions

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.

3. Problem statement

Consider the case of someone walking home while on an Internet-based audio call. 4G coverage in his area is good, but for cost reasons, he prefers to use wifi when available.

As he nears his house, the phone picks up a weak wifi signal - high packet loss, low bandwidth, but definitely present. Next to his front door, there is a big tree. As he walks behind the tree, the signal disappears. As he walks in the front door, the wifi signal is strong and stable.

What media should the call use while it is progressing through these scenarios?

4. Traditional ICE

The traditional ([RFC8863]) version of ICE can be briefly described as:

- * Generate a large number of candidate pairs
- * Try them in order until one of them works
- * Start using the working one (ICE controller decides)
- * Throw away all the other ones
- * When the chosen pair breaks, do an ICE restart and start over

There are extensions specified to this model; one of particular interest is Trickle ICE (RFC 8838), which allows adding more candidates after initialization, forming new sets of candidate pairs without an ICE restart.

5. NICER - the idea

The idea behind NICER is that rather than keeping a single candidate pair up, the ICE controller will form a list of candidate pairs it considers "potentially viable". The ICE controller will perform STUN Pings (bind requests) on these pairs to keep them alive and get some metrics on quality (RTT, packet loss).

When the ICE controller decides that one of these pairs is doing better than the currently active candidate pair, it will switch the active pair to this pair, and relegate the old pair to the "alternate" pool, informing the other party through a BIND request with the "nominated" flag set.

The ICE controller will still discard candidate pairs that never started working, and candidate pairs that have a high likelihood of being duplicates of other candidate pairs in the pool.

When new network interfaces come up, the ICE controller will use trickle-ICE to communicate with the other party and form new sets of candidate pairs; when interfaces go down, the ICE controller will switch to a still-working interface at once; ICE restart will only happen once all previously usable connections have failed.

It follows from the description above that NICER may need to add candidates at any time; the simplest approach compatible with standard ICE is to never send end-of-candidates, but more subtle approaches should be possible.

6. Standardization requirements

Most of the adaptations needed for NICER are within the ICE controller, and don't need to be standardized. However, there are a few parts that affect messages on the wire, and these call for some standardization effort - either by pushing through existing proposals that cover the need, or by standardizing new features.

These are:

- * Trickle ICE [RFC8838]
- * Continuous renomination

Continuous renomination means that for some subset of candidate pairs not selected, rather than discarding them as mandated by [RFC8863] section 8.3, they will be retained and be made available for selection by sending a check with the USE-CANDIDATE attribute on that candidate pair. One writeup for an extension that would permit this was draft-thatcher-ice-renomination (expired I-D).

With these features in place, NICER should be deployable against any system that implements these features.

7. Possible optimizations

In Google's experimentation with NICER, we have experimented with reducing the size of the ping - omitting known parameters and using shorter message-integrity functions. These optimizations may be interesting to standardize, but not essential for making NICER work.

8. Implementation issues

These are things for which standardization is not needed, but where implementors who want to use NICER need to be aware of them.

The definition of "better" is quite fluid and complex. The data available from a connection that is only occasionally pinged is also limited; for instance, bandwidth limitations can't be probed with just occasional probe packets (although guesses can be made). In particular, the ICE concept of "priority" (used to rank candidate pairs consistently at the ICE controller and controlled entities) is not useful for ranking the preferability of candidates for switching.

Managing power budgets on mobile devices can be challenging. In particular, pinging interfaces keeps radios alive and therefore consume power; when an interface has no reachable connections, one should avoid pinging it.

Given the dynamic nature of RF environments, occasional pinging runs the risk of decisions being taken on stale data, while frequent pings use battery and bandwidth; tuning these tradeoffs requires some attention while implementing.

9. Security Considerations

Keeping additional paths open increases the attack area for MITM attacks, naturally. So just as in the case of other TURN usages, it is important that the traffic sent over these TURN connections be authenticated and encrypted as appropriate.

Shortening the checksum will weaken the barrier to impersonation. This may not matter if the shortened checksum is only used on subsequent pings, not initial handshakes.

10. IANA Considerations

This document has no IANA actions.

11. Normative References

- [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/rfc/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/rfc/rfc8174>>.
- [RFC8838] Ivov, E., Uberti, J., and P. Saint-Andre, "Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol", RFC 8838, DOI 10.17487/RFC8838, January 2021, <<https://www.rfc-editor.org/rfc/rfc8838>>.
- [RFC8863] Holmberg, C. and J. Uberti, "Interactive Connectivity Establishment Patiently Awaiting Connectivity (ICE PAC)", RFC 8863, DOI 10.17487/RFC8863, January 2021, <<https://www.rfc-editor.org/rfc/rfc8863>>.

Acknowledgments

TODO acknowledge.

Authors' Addresses

Jonas Orelund
Google

Email: jonaso@google.com

Harald Alvestrand
Google

Email: hta@google.com

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: 16 July 2022

J. Zern
Google LLC
12 January 2022

WebP Image Format Media Type Registration
draft-zern-webp-06

Abstract

This document provides the Media Type Registration for the subtype image/webp.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 16 July 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
2. The 'image/webp' Media Type	3
2.1. Registration Details	3
3. Security Considerations	5
4. Interoperability Considerations	5
5. IANA Considerations	5
6. WebP Container Specification	5
6.1. Introduction	6
6.2. Terminology & Basics	6
6.3. RIFF File Format	7
6.4. WebP File Header	8
6.5. Simple File Format (Lossy)	8
6.6. Simple File Format (Lossless)	9
6.7. Extended File Format	10
6.7.1. Chunks	12
6.7.1.1. Animation	12
6.7.1.2. Alpha	15
6.7.1.3. Bitstream (VP8/VP8L)	18
6.7.1.4. Color profile	18
6.7.1.5. Metadata	19
6.7.1.6. Unknown Chunks	19
6.7.2. Assembling the Canvas from frames	20
6.7.3. Example File Layouts	21
7. Specification for WebP Lossless Bitstream	22
7.1. Abstract	22
7.2. Nomenclature	22
7.3. Introduction	24
7.4. RIFF Header	25
7.5. Transformations	25
7.5.1. Predictor Transform	26
7.5.2. Color Transform	29
7.5.3. Subtract Green Transform	31
7.5.4. Color Indexing Transform	32
7.6. Image Data	34
7.6.1. Roles of Image Data	34
7.6.2. Encoding of Image data	34
7.6.2.1. Huffman Coded Literals	35
7.6.2.2. LZ77 Backward Reference	35
7.6.2.3. Color Cache Coding	37
7.7. Entropy Code	38
7.7.1. Overview	38
7.7.2. Details	38
7.7.2.1. Decoding of Meta Huffman Codes	39
7.7.2.2. Decoding Entropy-coded Image Data	40
7.8. Overall Structure of the Format	43
7.8.1. Basic Structure	43

7.8.2. Structure of Transforms	43
7.8.3. Structure of the Image Data	43
8. References	44
8.1. Normative References	44
8.2. Informative References	45
Author's Address	46

1. Introduction

This document provides references for the WebP image format and considerations for its use across platforms.

WebP is a Resource Interchange File Format (RIFF) [riff-spec] based image file format (Section 6) which supports lossless and lossy compression as well as alpha (transparency) and animation. It covers use cases similar to JPEG [jpeg-spec], PNG [RFC2083] and the Graphics Interchange Format (GIF) [gif-spec].

WebP consists of two compression algorithms used to reduce the size of image pixel data, including alpha (transparency) information. Lossy compression is achieved using VP8 intra-frame encoding [RFC6386]. The lossless algorithm (Section 7) stores and restores the pixel values exactly, including the color values for zero alpha pixels. The format uses subresolution images, recursively embedded into the format itself, for storing statistical data about the images, such as the used entropy codes, spatial predictors, color space conversion, and color table. LZ77 [lz77], Huffman coding [huffman], and a color cache are used for compression of the bulk data.

2. The 'image/webp' Media Type

This section contains the media type registration details as per [RFC6838].

2.1. Registration Details

Type name: image

Subtype name: webp

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: Binary. The Base64 encoding [RFC4648] should be used on transports that cannot accommodate binary data directly.

Security considerations: See Section 3.

Interoperability considerations: See Section 4.

Published specification: [webp-riff-src]

Applications that use this media type: Applications that are used to display and process images, especially when smaller image file sizes are important.

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): The first 4 bytes are 0x52, 0x49, 0x46, 0x46 ('RIFF'), followed by 4 bytes for the RIFF chunk size. The next 7 bytes are 0x57, 0x45, 0x42, 0x50, 0x56, 0x50, 0x38 ('WEBPVP8').

File extension(s): webp

Apple Uniform Type Identifier: org.webmproject.webp conforms to public.image

Object Identifiers: N/A

Person & email address to contact for further information:

Name: James Zern

Email: jzern@google.com

Intended usage: COMMON

Restrictions on usage: N/A

Author:

Name: James Zern

Email: jzern@google.com

Change controller:

Name: James Zern

Email: jzern@google.com

Name: Pascal Massimino

Email: pascal.massimino@gmail.com

Name: WebM Project

Email: webmaster@webmproject.org

Provisional registration? (standards tree only): N/A

3. Security Considerations

Security risks are similar to other media content and may include integer overflows, out-of-bounds reads and writes to both heap and stack, uninitialized data usage, null pointer references, resource (disk, memory) exhaustion and extended resource usage (long running time) as part of the demuxing and decoding process. These may cause information leakage (memory layout and contents) or crashes and thereby denial of service to an application using the format [cve.mitre.org-libwebp] [crbug-security].

The format does not employ "active content", but does allow metadata ([XMP], [Exif]) and custom chunks to be embedded in a file. Applications that interpret these chunks may be subject to security considerations for those formats.

4. Interoperability Considerations

The format is defined using little-endian byte ordering (see Section 3.1 of [RFC2781]), but demuxing and decoding are possible on platforms using a different ordering with the appropriate conversion. The container is RIFF-based and allows extension via user defined chunks, but nothing beyond the chunks defined by the container format (Section 6) are required for decoding of the image. These have been finalized, but were extended in the format's early stages so some older readers may not support lossless or animated image decoding.

5. IANA Considerations

IANA has updated the "Image Media Types" registry [IANA-Media-Types] to include 'image/webp' as described in Section 2.

6. WebP Container Specification

Note this section is based on the documentation in the libwebp source repository [webp-riff-src] at the time of writing.

6.1. Introduction

WebP is an image format that uses either (i) the VP8 intra-frame encoding [RFC6386] to compress image data in a lossy way, or (ii) the WebP lossless encoding (Section 7). These encoding schemes should make it more efficient than currently used formats. It is optimized for fast image transfer over the network (e.g., for websites). The WebP format has feature parity (color profile, metadata, animation etc) with other formats as well. This section describes the structure of a WebP file.

The WebP container (i.e., RIFF container for WebP) allows feature support over and above the basic use case of WebP (i.e., a file containing a single image encoded as a VP8 key frame). The WebP container provides additional support for:

- * **Lossless compression.** An image can be losslessly compressed, using the WebP Lossless Format.
- * **Metadata.** An image may have metadata stored in [Exif] or [XMP] formats.
- * **Transparency.** An image may have transparency, i.e., an alpha channel.
- * **Color Profile.** An image may have an embedded ICC profile [ICC].
- * **Animation.** An image may have multiple frames with pauses between them, making it an animation.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Bit numbering in chunk diagrams starts at 0 for the most significant bit ('MSB 0') as described in [RFC1166].

6.2. Terminology & Basics

A WebP file contains either a still image (i.e., an encoded matrix of pixels) or an animation (Section 6.7.1.1). Optionally, it can also contain transparency information, color profile and metadata. In case we need to refer only to the matrix of pixels, we will call it the `_canvas_` of the image.

Below are additional terms used throughout this document:

Reader/Writer

Code that reads WebP files is referred to as a `_reader_`, while code that writes them is referred to as a `_writer_`.

uint16

A 16-bit, little-endian, unsigned integer.

uint24

A 24-bit, little-endian, unsigned integer.

uint32

A 32-bit, little-endian, unsigned integer.

FourCC

A FourCC (four-character code) is a uint32 created by concatenating four ASCII characters in little-endian order.

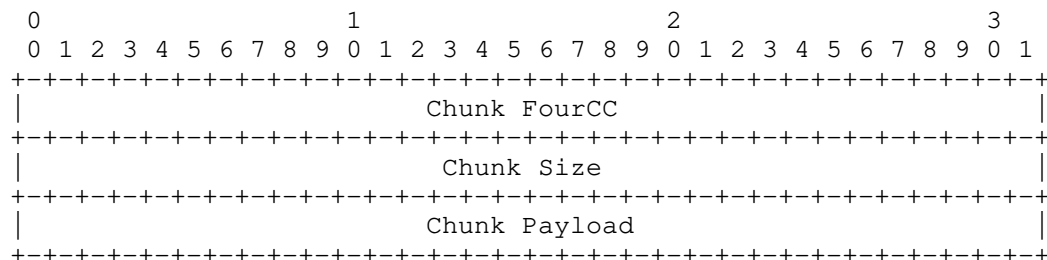
1-based

An unsigned integer field storing values offset by -1. e.g., Such a field would store value `_25_` as `_24_`.

6.3. RIFF File Format

The WebP file format is based on the RIFF [riff-spec] (Resource Interchange File Format) document format.

The basic element of a RIFF file is a `_chunk_`. It consists of:



Chunk FourCC: 32 bits

ASCII four-character code used for chunk identification.

Chunk Size: 32 bits (`_uint32_`)

The size of the chunk not including this field, the chunk identifier or padding.

Chunk Payload: `_Chunk Size_` bytes

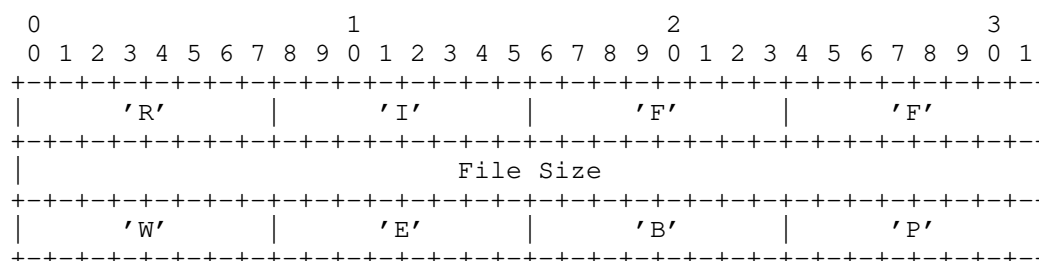
The data payload. If `_Chunk Size_` is odd, a single padding byte -- that SHOULD be 0 -- is added.

ChunkHeader('ABCD')

This is used to describe the `_FourCC_` and `_Chunk Size_` header of individual chunks, where 'ABCD' is the FourCC for the chunk. This element's size is 8 bytes.

Note: RIFF has a convention that all-uppercase chunk FourCCs are standard chunks that apply to any RIFF file format, while FourCCs specific to a file format are all lowercase. WebP does not follow this convention.

6.4. WebP File Header



'RIFF': 32 bits

The ASCII characters 'R' 'I' 'F' 'F'.

File Size: 32 bits (`uint32_t`)

The size of the file in bytes starting at offset 8. The maximum value of this field is 2^{32} minus 10 bytes and thus the size of the whole file is at most 4GiB minus 2 bytes.

'WEBP': 32 bits

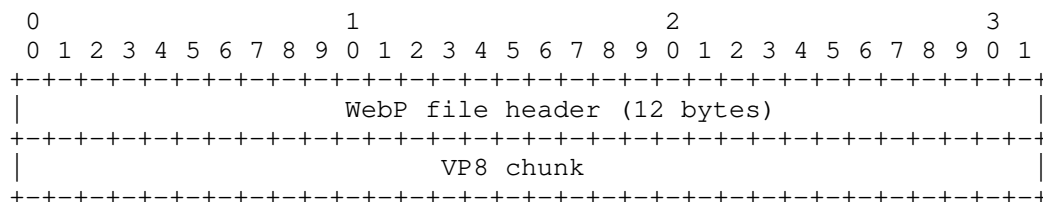
The ASCII characters 'W' 'E' 'B' 'P'.

A WebP file MUST begin with a RIFF header with the FourCC 'WEBP'. The file size in the header is the total size of the chunks that follow plus 4 bytes for the 'WEBP' FourCC. The file SHOULD NOT contain anything after it. As the size of any chunk is even, the size given by the RIFF header is also even. The contents of individual chunks will be described in the following sections.

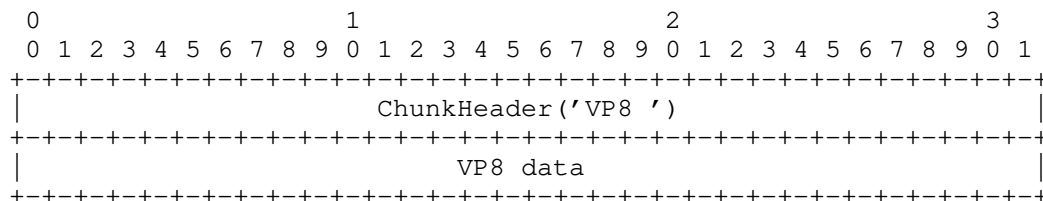
6.5. Simple File Format (Lossy)

This layout SHOULD be used if the image requires lossy encoding and does not require transparency or other advanced features provided by the extended format. Files with this layout are smaller and supported by older software.

Simple WebP (lossy) file format:



VP8 chunk:



```
VP8 data: _Chunk Size_ bytes
          VP8 bitstream data.
```

The VP8 bitstream format specification is described by [RFC6386]. Note that the VP8 frame header contains the VP8 frame width and height. That is assumed to be the width and height of the canvas.

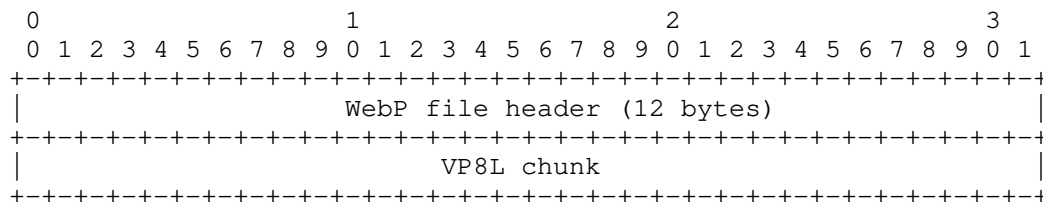
The VP8 specification describes how to decode the image into Y'CbCr format. To convert to RGB, Rec. 601 [rec601] SHOULD be used.

6.6. Simple File Format (Lossless)

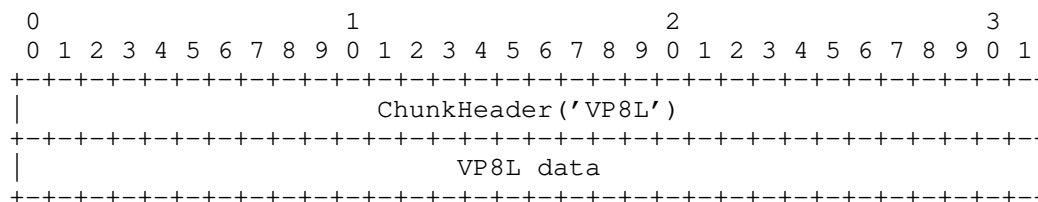
*Note: Older readers may not support files using the lossless format.

This layout SHOULD be used if the image requires lossless encoding (with an optional transparency channel) and does not require advanced features provided by the extended format.

Simple WebP (lossless) file format:



VP8L chunk:



VP8L data: Chunk Size bytes
 VP8L bitstream data.

The specification of the VP8L bitstream can be found in Section 7. Note that the VP8L header contains the VP8L image width and height. That is assumed to be the width and height of the canvas.

6.7. Extended File Format

Note: Older readers may not support files using the extended format.

An extended format file consists of:

- * A 'VP8X' chunk with information about features used in the file.
- * An optional 'ICCP' chunk with color profile.
- * An optional 'ANIM' chunk with animation control data.
- * Image data.
- * An optional 'EXIF' chunk with Exif metadata.
- * An optional 'XMP' chunk with XMP metadata.
- * An optional list of unknown chunks (Section 6.7.1.6).

For a still image, the image data consists of a single frame, which is made up of:

- * An optional alpha subchunk (Section 6.7.1.2).
- * A bitstream subchunk (Section 6.7.1.3).

For an animated image, the image data consists of multiple frames. More details about frames can be found in Section 6.7.1.1.

All chunks SHOULD be placed in the same order as listed above. If a chunk appears in the wrong place, the file is invalid, but readers MAY parse the file, ignoring the chunks that come too late.

Rationale: Setting the order of chunks should allow quicker file parsing. For example, if an 'ALPH' chunk does not appear in its required position, a decoder can choose to stop searching for it. The rule of ignoring late chunks should make programs that need to do a full search give the same results as the ones stopping early.

Extended WebP file header:

```

0                                     1                                     2                                     3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               WebP file header (12 bytes)                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               ChunkHeader('VP8X')                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Rsv|I|L|E|X|A|R|                               Reserved                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               Canvas Width Minus One                               |                               ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
... Canvas Height Minus One |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

```

Reserved (Rsv): 2 bits
SHOULD be 0.

ICC profile (I): 1 bit
Set if the file contains an ICC profile.

Alpha (L): 1 bit
Set if any of the frames of the image contain transparency information ("alpha").

```
Exif metadata (E): 1 bit
    Set if the file contains Exif metadata.
```

XMP metadata (X): 1 bit
Set if the file contains XMP metadata.

```

Animation (A): 1 bit
    Set if this is an animated image. Data in 'ANIM' and 'ANMF'
    chunks should be used to control the animation.

```

Reserved (R): 1 bit
SHOULD be 0.

Reserved: 24 bits
SHOULD be 0.

Canvas Width Minus One: 24 bits
1-based width of the canvas in pixels. The actual canvas width is 1 + Canvas Width Minus One

Canvas Height Minus One: 24 bits
1-based height of the canvas in pixels. The actual canvas height is 1 + Canvas Height Minus One

The product of _Canvas Width_ and _Canvas Height_ MUST be at most $2^{32} - 1$.

Future specifications MAY add more fields.

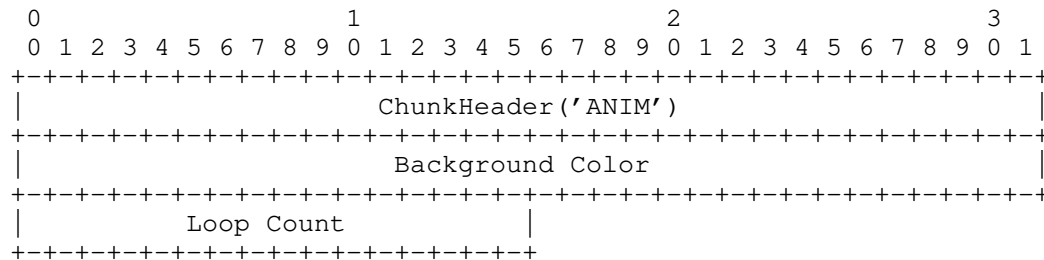
6.7.1. Chunks

6.7.1.1. Animation

An animation is controlled by ANIM and ANMF chunks.

ANIM Chunk:

For an animated image, this chunk contains the _global parameters_ of the animation.



Background Color: 32 bits (_uint32_)

The default background color of the canvas in [Blue, Green, Red, Alpha] byte order. This color MAY be used to fill the unused space on the canvas around the frames, as well as the transparent pixels of the first frame. Background color is also used when disposal method is 1.

Note:

- * Background color MAY contain a transparency value (alpha), even if the `_Alpha_` flag in VP8X chunk (Section 6.7, Paragraph 9) is unset.
- * Viewer applications SHOULD treat the background color value as a hint, and are not required to use it.
- * The canvas is cleared at the start of each loop. The background color MAY be used to achieve this.

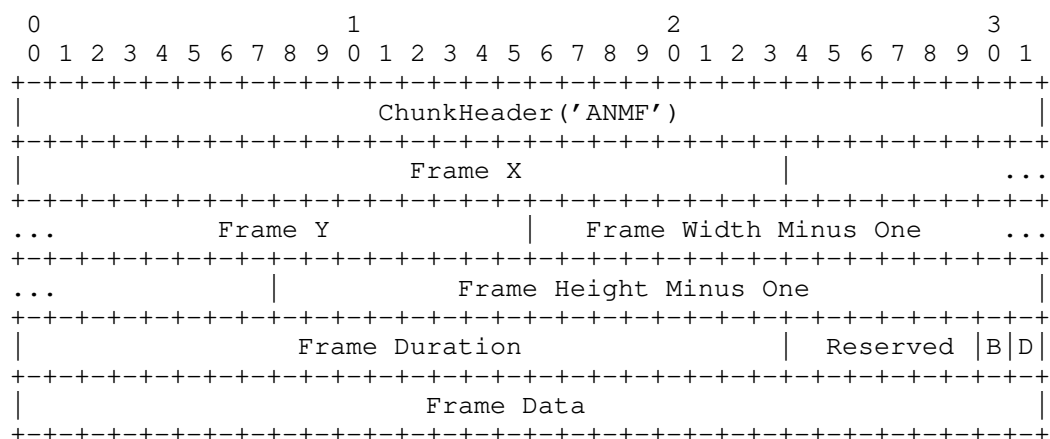
Loop Count: 16 bits (`_uint16_`)

The number of times to loop the animation. 0 means infinitely.

This chunk MUST appear if the `_Animation_` flag in the VP8X chunk is set. If the `_Animation_` flag is not set and this chunk is present, it SHOULD be ignored.

ANMF chunk:

For animated images, this chunk contains information about a `_single_` frame. If the `_Animation_` flag is not set, then this chunk SHOULD NOT be present.



Frame X: 24 bits (`_uint24_`)

The X coordinate of the upper left corner of the frame is
Frame X * 2

Frame Y: 24 bits (`_uint24_`)

The Y coordinate of the upper left corner of the frame is
Frame Y * 2

Frame Width Minus One: 24 bits (`_uint24_`)

The `_1-based_` width of the frame. The frame width is 1 +
Frame Width Minus One

Frame Height Minus One: 24 bits (`_uint24_`)

The `_1-based_` height of the frame. The frame height is 1 +
Frame Height Minus One

Frame Duration: 24 bits (`_uint24_`)

The time to wait before displaying the next frame, in 1
millisecond units. Note the interpretation of frame duration
of 0 (and often ≤ 10) is implementation defined. Many tools
and browsers assign a minimum duration similar to GIF.

Reserved: 6 bits

SHOULD be 0.

Blending method (B): 1 bit

Indicates how transparent pixels of `_the current frame_` are
to be blended with corresponding pixels of the previous
canvas:

- * 0: Use alpha blending. After disposing of the previous
frame, render the current frame on the canvas using
alpha-blending (Section 6.7.1.1, Paragraph 10, Item
16.4.2). If the current frame does not have an alpha
channel, assume alpha value of 255, effectively replacing
the rectangle.
- * 1: Do not blend. After disposing of the previous frame,
render the current frame on the canvas by overwriting the
rectangle covered by the current frame.

Disposal method (D): 1 bit

Indicates how `_the current frame_` is to be treated after it
has been displayed (before rendering the next frame) on the
canvas:

- * 0: Do not dispose. Leave the canvas as is.
- * 1: Dispose to background color. Fill the `_rectangle_` on
the canvas covered by the `_current frame_` with background
color specified in the ANIM chunk (Section 6.7.1.1,
Paragraph 2).

Notes:

- * The frame disposal only applies to the `_frame rectangle_`, that is, the rectangle defined by `_Frame X_`, `_Frame Y_`, `_frame width_` and `_frame height_`. It may or may not cover the whole canvas.
- * Alpha-blending:

Given that each of the R, G, B and A channels is 8-bit, and the RGB channels are `_not premultiplied_` by alpha, the formula for blending 'dst' onto 'src' is:

```
blend.A = src.A + dst.A * (1 - src.A / 255)
if blend.A = 0 then
    blend.RGB = 0
else
    blend.RGB = (src.RGB * src.A +
                 dst.RGB * dst.A * (1 - src.A / 255)) / blend.A
```
- * Alpha-blending SHOULD be done in linear color space, by taking into account the color profile (Section 6.7.1.4) of the image. If the color profile is not present, sRGB is to be assumed. (Note that sRGB also needs to be linearized due to a gamma of ~2.2).

Frame Data: `_Chunk Size_` - 16 bytes
Consists of:

- * An optional alpha subchunk (Section 6.7.1.2) for the frame.
- * A bitstream subchunk (Section 6.7.1.3) for the frame.
- * An optional list of unknown chunks (Section 6.7.1.6).

Note: The 'ANMF' payload, `_Frame Data_` above, consists of individual `_padded_` chunks as described by the RIFF file format (Section 6.3).

6.7.1.2. Alpha

```

      0               1               2               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+
|                                     ChunkHeader('ALPH')                                     |
+-----+-----+-----+-----+
|Rsv| P | F | C | Alpha Bitstream... |
+-----+-----+-----+-----+
```

Reserved (Rsv): 2 bits
SHOULD be 0.

Pre-processing (P): 2 bits
These INFORMATIVE bits are used to signal the pre-processing that has been performed during compression. The decoder can use this information to e.g. dither the values or smooth the gradients prior to display.

- * 0: no pre-processing
- * 1: level reduction

Filtering method (F): 2 bits
The filtering method used:

- * 0: None.
- * 1: Horizontal filter.
- * 2: Vertical filter.
- * 3: Gradient filter.

For each pixel, filtering is performed using the following calculations. Assume the alpha values surrounding the current X position are labeled as:

```
  C | B |  
---+---+  
  A | X |
```

We seek to compute the alpha value at position X. First, a prediction is made depending on the filtering method:

- * Method 0: predictor = 0
- * Method 1: predictor = A
- * Method 2: predictor = B
- * Method 3: predictor = clip(A + B - C)

where clip(v) is equal to:

- * 0 if v < 0
- * 255 if v > 255

* v otherwise

The final value is derived by adding the decompressed value X to the predictor and using modulo-256 arithmetic to wrap the [256-511] range into the [0-255] one:

$\alpha = (\text{predictor} + X) \% 256$

There are special cases for left-most and top-most pixel positions:

- * Top-left value at location (0,0) uses 0 as predictor value. Otherwise,
- * For horizontal or gradient filtering methods, the left-most pixels at location (0, y) are predicted using the location (0, y-1) just above.
- * For vertical or gradient filtering methods, the top-most pixels at location (x, 0) are predicted using the location (x-1, 0) on the left.

Decoders are not required to use this information in any specified way.

Compression method (C): 2 bits

The compression method used:

- * 0: No compression.
- * 1: Compressed using the WebP lossless format.

Alpha bitstream: Chunk Size - 1 bytes

Encoded alpha bitstream.

This optional chunk contains encoded alpha data for this frame. A frame containing a 'VP8L' chunk SHOULD NOT contain this chunk.

Rationale: The transparency information is already part of the 'VP8L' chunk.

The alpha channel data is stored as uncompressed raw data (when compression method is '0') or compressed using the lossless format (when the compression method is '1').

- * Raw data: consists of a byte sequence of length width * height, containing all the 8-bit transparency values in scan order.

- * Lossless format compression: the byte sequence is a compressed image-stream (as described in Section 7) of implicit dimension width x height. That is, this image-stream does NOT contain any headers describing the image dimension.

Rationale: the dimension is already known from other sources, so storing it again would be redundant and error-prone.

Once the image-stream is decoded into ARGB color values, following the process described in the lossless format specification, the transparency information must be extracted from the *green* channel of the ARGB quadruplet.

```
*Rationale:* the green channel is allowed extra transformation
steps in the specification -- unlike the other channels -- that
can improve compression.
```

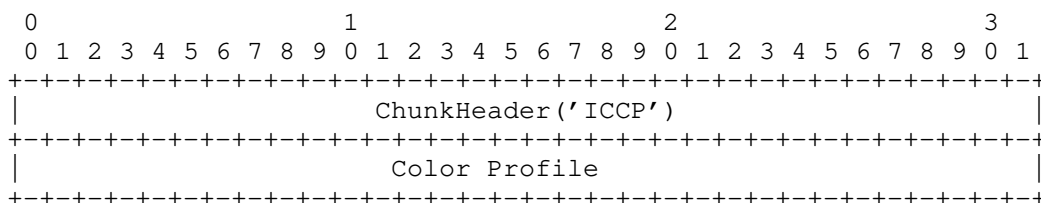
6.7.1.3. Bitstream (VP8/VP8L)

This chunk contains compressed bitstream data for a single frame.

A bitstream chunk may be either (i) a VP8 chunk, using "VP8 " (note the significant fourth-character space) as its tag_or_ (ii) a VP8L chunk, using "VP8L" as its tag.

The formats of VP8 and VP8L chunks are as described in Section 6.5 and Section 6.6 respectively.

6.7.1.4. Color profile



Color Profile: _Chunk Size_ bytes
ICC profile.

This chunk MUST appear before the image data.

There SHOULD be at most one such chunk. If there are more such chunks, readers MAY ignore all except the first one. See the ICC Specification [ICC] for details.

If this chunk is not present, sRGB SHOULD be assumed.

6.7.1.5. Metadata

Metadata can be stored in 'EXIF' or 'XMP' chunks.

There SHOULD be at most one chunk of each type ('EXIF' and 'XMP'). If there are more such chunks, readers MAY ignore all except the first one. Also, a file may possibly contain both 'EXIF' and 'XMP' chunks.

The chunks are defined as follows:

EXIF chunk:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     ChunkHeader('EXIF')                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Exif Metadata                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Exif Metadata: Chunk Size bytes
image metadata in [Exif] format.

XMP chunk:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     ChunkHeader('XMP ')                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     XMP Metadata                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

XMP Metadata: Chunk Size bytes
image metadata in [XMP] format.

Additional guidance about handling metadata can be found in the Metadata Working Group's Guidelines for Handling Metadata [mwg].

6.7.1.6. Unknown Chunks

A RIFF chunk (described in Section 6.2.) whose chunk tag is different from any of the chunks described in this document, is considered an unknown chunk.

Rationale: Allowing unknown chunks gives a provision for future extension of the format, and also allows storage of any application-specific data.

A file MAY contain unknown chunks:

- * At the end of the file as described in Section 6.7, Paragraph 9.
- * At the end of ANMF chunks as described in Section 6.7.1.1.

Readers SHOULD ignore these chunks. Writers SHOULD preserve them in their original order (unless they specifically intend to modify these chunks).

6.7.2. Assembling the Canvas from frames

Here we provide an overview of how a reader should assemble a canvas in the case of an animated image. The notation `_VP8X.field_` means the field in the 'VP8X' chunk with the same description.

Displaying an `_animated image_` canvas MUST be equivalent to the following pseudocode:

```

assert VP8X.flags.hasAnimation
canvas <- new image of size VP8X.canvasWidth x VP8X.canvasHeight with
    background color ANIM.background_color.
loop_count <- ANIM.loopCount
dispose_method <- ANIM.disposeMethod
if loop_count == 0:
    loop_count = inf
frame_params <- nil
assert next chunk in image_data is ANMF
for loop = 0..loop_count - 1
    clear canvas to ANIM.background_color or application defined color
    until eof or non-ANMF chunk
        frame_params.frameX = Frame X
        frame_params.frameY = Frame Y
        frame_params.frameWidth = Frame Width Minus One + 1
        frame_params.frameHeight = Frame Height Minus One + 1
        frame_params.frameDuration = Frame Duration
        frame_right = frame_params.frameX + frame_params.frameWidth
        frame_bottom = frame_params.frameY + frame_params.frameHeight
        assert VP8X.canvasWidth >= frame_right
        assert VP8X.canvasHeight >= frame_bottom
        for subchunk in 'Frame Data':
            if subchunk.tag == "ALPH":
                assert alpha subchunks not found in 'Frame Data' earlier
                frame_params.alpha = alpha_data
            else if subchunk.tag == "VP8 " OR subchunk.tag == "VP8L":
                assert bitstream subchunks not found in 'Frame Data' earlier
                frame_params.bitstream = bitstream_data
        render frame with frame_params.alpha and frame_params.bitstream on
        canvas with top-left corner at (frame_params.frameX,
            frame_params.frameY), using dispose method dispose_method.
        canvas contains the decoded image.
        Show the contents of the canvas for frame_params.frameDuration * 1ms.

```

6.7.3. Example File Layouts

A lossy encoded image with alpha may look as follows:

```

RIFF/WEBP
+- VP8X (descriptions of features used)
+- ALPH (alpha bitstream)
+- VP8 (bitstream)

```

A losslessly encoded image may look as follows:

```
RIFF/WEBP
+- VP8X (descriptions of features used)
+- XYZW (unknown chunk)
+- VP8L (lossless bitstream)
```

A lossless image with ICC profile and XMP metadata may look as follows:

```
RIFF/WEBP
+- VP8X (descriptions of features used)
+- ICCP (color profile)
+- VP8L (lossless bitstream)
+- XMP (metadata)
```

An animated image with Exif metadata may look as follows:

```
RIFF/WEBP
+- VP8X (descriptions of features used)
+- ANIM (global animation parameters)
+- ANMF (frame1 parameters + data)
+- ANMF (frame2 parameters + data)
+- ANMF (frame3 parameters + data)
+- ANMF (frame4 parameters + data)
+- EXIF (metadata)
```

7. Specification for WebP Lossless Bitstream

Note this section is based on the documentation in the libwebp source repository [webp-lossless-src] at the time of writing.

7.1. Abstract

WebP lossless is an image format for lossless compression of ARGB images. The lossless format stores and restores the pixel values exactly, including the color values for zero alpha pixels. The format uses subresolution images, recursively embedded into the format itself, for storing statistical data about the images, such as the used entropy codes, spatial predictors, color space conversion, and color table. LZ77, Huffman coding, and a color cache are used for compression of the bulk data. Decoding speeds faster than PNG have been demonstrated, as well as 25% denser compression than can be achieved using today's PNG format.

7.2. Nomenclature

ARGB

A pixel value consisting of alpha, red, green, and blue values.

ARGB image

A two-dimensional array containing ARGB pixels.

color cache

A small hash-addressed array to store recently used colors, to be able to recall them with shorter codes.

color indexing image

A one-dimensional image of colors that can be indexed using a small integer (up to 256 within WebP lossless).

color transform image

A two-dimensional subresolution image containing data about correlations of color components.

distance mapping

Changes LZ77 distances to have the smallest values for pixels in 2D proximity.

entropy image

A two-dimensional subresolution image indicating which entropy coding should be used in a respective square in the image, i.e., each pixel is a meta Huffman code.

Huffman code

A classic way to do entropy coding where a smaller number of bits are used for more frequent codes.

LZ77

Dictionary-based sliding window compression algorithm that either emits symbols or describes them as sequences of past symbols.

meta Huffman code

A small integer (up to 16 bits) that indexes an element in the meta Huffman table.

predictor image

A two-dimensional subresolution image indicating which spatial predictor is used for a particular square in the image.

prefix coding

A way to entropy code larger integers that codes a few bits of the integer using an entropy code and codifies the remaining bits raw. This allows for the descriptions of the entropy codes to remain relatively small even when the range of symbols is large.

scan-line order

A processing order of pixels, left-to-right, top-to-bottom, starting from the left-hand-top pixel, proceeding to the right. Once a row is completed, continue from the left-hand column of the next row.

7.3. Introduction

This document describes the compressed data representation of a WebP lossless image. It is intended as a detailed reference for WebP lossless encoder and decoder implementation.

In this document, we extensively use C programming language syntax to describe the bitstream, and assume the existence of a function for reading bits, `ReadBits(n)`. The bytes are read in the natural order of the stream containing them, and bits of each byte are read in least-significant-bit-first order. When multiple bits are read at the same time, the integer is constructed from the original data in the original order. The most significant bits of the returned integer are also the most significant bits of the original data. Thus the statement

```
b = ReadBits(2);
```

is equivalent with the two statements below:

```
b = ReadBits(1);
b |= ReadBits(1) << 1;
```

We assume that each color component (e.g. alpha, red, blue and green) is represented using an 8-bit byte. We define the corresponding type as `uint8`. A whole ARGB pixel is represented by a type called `uint32`, an unsigned integer consisting of 32 bits. In the code showing the behavior of the transformations, alpha value is codified in bits 31..24, red in bits 23..16, green in bits 15..8 and blue in bits 7..0, but implementations of the format are free to use another representation internally.

Broadly, a WebP lossless image contains header data, transform information and actual image data. Headers contain width and height of the image. A WebP lossless image can go through four different types of transformation before being entropy encoded. The transform information in the bitstream contains the data required to apply the respective inverse transforms.

7.4. RIFF Header

The beginning of the header has the RIFF container. This consists of the following 21 bytes:

1. String "RIFF"
2. A little-endian 32 bit value of the block length, the whole size of the block controlled by the RIFF header. Normally this equals the payload size (file size minus 8 bytes: 4 bytes for the 'RIFF' identifier and 4 bytes for storing the value itself).
3. String "WEBP" (RIFF container name).
4. String "VP8L" (chunk tag for lossless encoded image data).
5. A little-endian 32-bit value of the number of bytes in the lossless stream.
6. One byte signature 0x2f.

The first 28 bits of the bitstream specify the width and height of the image. Width and height are decoded as 14-bit integers as follows:

```
int image_width = ReadBits(14) + 1;
int image_height = ReadBits(14) + 1;
```

The 14-bit dynamics for image size limit the maximum size of a WebP lossless image to 16384x16384 pixels.

The `alpha_is_used` bit is a hint only, and should not impact decoding. It should be set to 0 when all alpha values are 255 in the picture, and 1 otherwise.

```
int alpha_is_used = ReadBits(1);
```

The `version_number` is a 3 bit code that must be set to 0. Any other value should be treated as an error.

```
int version_number = ReadBits(3);
```

7.5. Transformations

Transformations are reversible manipulations of the image data that can reduce the remaining symbolic entropy by modeling spatial and color correlations. Transformations can make the final compression more dense.

An image can go through four types of transformation. A 1 bit indicates the presence of a transform. Each transform is allowed to be used only once. The transformations are used only for the main level ARGB image: the subresolution images have no transforms, not even the 0 bit indicating the end-of-transforms.

Typically an encoder would use these transforms to reduce the Shannon entropy in the residual image. Also, the transform data can be decided based on entropy minimization.

```
while (ReadBits(1)) { // Transform present.
    // Decode transform type.
    enum TransformType transform_type = ReadBits(2);
    // Decode transform data.
    ...
}

// Decode actual image data.
```

If a transform is present then the next two bits specify the transform type. There are four types of transforms.

```
enum TransformType {
    PREDICTOR_TRANSFORM          = 0,
    COLOR_TRANSFORM              = 1,
    SUBTRACT_GREEN               = 2,
    COLOR_INDEXING_TRANSFORM     = 3,
};
```

The transform type is followed by the transform data. Transform data contains the information required to apply the inverse transform and depends on the transform type. Next we describe the transform data for different types.

7.5.1. Predictor Transform

The predictor transform can be used to reduce entropy by exploiting the fact that neighboring pixels are often correlated. In the predictor transform, the current pixel value is predicted from the pixels already decoded (in scan-line order) and only the residual value (actual - predicted) is encoded. The `_prediction mode_` determines the type of prediction to use. We divide the image into squares and all the pixels in a square use same prediction mode.

The first 3 bits of prediction data define the block width and height in number of bits. The number of block columns, `block_xsize`, is used in indexing two-dimensionally.

```

int size_bits = ReadBits(3) + 2;
int block_width = (1 << size_bits);
int block_height = (1 << size_bits);
#define DIV_ROUND_UP(num, den) ((num) + (den) - 1) / (den)
int block_xsize = DIV_ROUND_UP(image_width, 1 << size_bits);

```

The transform data contains the prediction mode for each block of the image. All the `block_width * block_height` pixels of a block use same prediction mode. The prediction modes are treated as pixels of an image and encoded using the same techniques described in Section 7.6.

For a pixel `_x, _y`, one can compute the respective filter block address by:

```

int block_index = (y >> size_bits) * block_xsize +
                  (x >> size_bits);

```

There are 14 different prediction modes. In each prediction mode, the current pixel value is predicted from one or more neighboring pixels whose values are already known.

We choose the neighboring pixels (TL, T, TR, and L) of the current pixel (P) as follows:

O	O	O	O	O	O	O	O	O	O	O
O	O	O	O	O	O	O	O	O	O	O
O	O	O	O	TL	T	TR	O	O	O	O
O	O	O	O	L	P	X	X	X	X	X
X	X	X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X	X	X

where TL means top-left, T top, TR top-right, L left pixel. At the time of predicting a value for P, all pixels O, TL, T, TR and L have been already processed, and pixel P and all pixels X are unknown.

Given the above neighboring pixels, the different prediction modes are defined as follows.

Mode	Predicted value of each channel of the current pixel
0	0xff000000 (represents solid black color in ARGB)
1	L
2	T
3	TR
4	TL
5	Average2(Average2(L, TR), T)
6	Average2(L, TL)
7	Average2(L, T)
8	Average2(TL, T)
9	Average2(T, TR)
10	Average2(Average2(L, TL), Average2(T, TR))
11	Select(L, T, TL)
12	ClampAddSubtractFull(L, T, TL)
13	ClampAddSubtractHalf(Average2(L, T), TL)

Average2 is defined as follows for each ARGB component:

```
uint8 Average2(uint8 a, uint8 b) {
    return (a + b) / 2;
}
```

The Select predictor is defined as follows:

```
uint32 Select(uint32 L, uint32 T, uint32 TL) {
    // L = left pixel, T = top pixel, TL = top left pixel.

    // ARGB component estimates for prediction.
    int pAlpha = ALPHA(L) + ALPHA(T) - ALPHA(TL);
    int pRed = RED(L) + RED(T) - RED(TL);
    int pGreen = GREEN(L) + GREEN(T) - GREEN(TL);
    int pBlue = BLUE(L) + BLUE(T) - BLUE(TL);

    // Manhattan distances to estimates for left and top pixels.
    int pL = abs(pAlpha - ALPHA(L)) + abs(pRed - RED(L)) +
             abs(pGreen - GREEN(L)) + abs(pBlue - BLUE(L));
    int pT = abs(pAlpha - ALPHA(T)) + abs(pRed - RED(T)) +
             abs(pGreen - GREEN(T)) + abs(pBlue - BLUE(T));

    // Return either left or top, the one closer to the prediction.
    if (pL < pT) {
        return L;
    } else {
        return T;
    }
}
```

The functions `ClampAddSubtractFull` and `ClampAddSubtractHalf` are performed for each ARGB component as follows:

```
// Clamp the input value between 0 and 255.
int Clamp(int a) {
    return (a < 0) ? 0 : (a > 255) ? 255 : a;
}

int ClampAddSubtractFull(int a, int b, int c) {
    return Clamp(a + b - c);
}

int ClampAddSubtractHalf(int a, int b) {
    return Clamp(a + (a - b) / 2);
}
```

There are special handling rules for some border pixels. If there is a prediction transform, regardless of the mode [0..13] for these pixels, the predicted value for the left-topmost pixel of the image is 0xff000000, L-pixel for all pixels on the top row, and T-pixel for all pixels on the leftmost column.

Addressing the TR-pixel for pixels on the rightmost column is exceptional. The pixels on the rightmost column are predicted by using the modes [0..13] just like pixels not on border, but by using the leftmost pixel on the same row as the current TR-pixel. The TR-pixel offset in memory is the same for border and non-border pixels.

7.5.2. Color Transform

The goal of the color transform is to decorrelate the R, G and B values of each pixel. Color transform keeps the green (G) value as it is, transforms red (R) based on green and transforms blue (B) based on green and then based on red.

As is the case for the predictor transform, first the image is divided into blocks and the same transform mode is used for all the pixels in a block. For each block there are three types of color transform elements.

```
typedef struct {
    uint8 green_to_red;
    uint8 green_to_blue;
    uint8 red_to_blue;
} ColorTransformElement;
```

The actual color transformation is done by defining a color transform delta. The color transform delta depends on the `ColorTransformElement`, which is the same for all the pixels in a particular block. The delta is added during color transform. The inverse color transform then is just subtracting those deltas.

The color transform function is defined as follows:

```
void ColorTransform(uint8 red, uint8 blue, uint8 green,
                   ColorTransformElement *trans,
                   uint8 *new_red, uint8 *new_blue) {
    // Transformed values of red and blue components
    uint32 tmp_red = red;
    uint32 tmp_blue = blue;

    // Applying transform is just adding the transform deltas
    tmp_red += ColorTransformDelta(trans->green_to_red, green);
    tmp_blue += ColorTransformDelta(trans->green_to_blue, green);
    tmp_blue += ColorTransformDelta(trans->red_to_blue, red);

    *new_red = tmp_red & 0xff;
    *new_blue = tmp_blue & 0xff;
}
```

`ColorTransformDelta` is computed using a signed 8-bit integer representing a 3.5-fixed-point number, and a signed 8-bit RGB color channel (c) [-128..127] and is defined as follows:

```
int8 ColorTransformDelta(int8 t, int8 c) {
    return (t * c) >> 5;
}
```

A conversion from the 8-bit unsigned representation (`uint8`) to the 8-bit signed one (`int8`) is required before calling `ColorTransformDelta()`. It should be performed using 8-bit two's complement (that is: `uint8` range [128-255] is mapped to the [-128, -1] range of its converted `int8` value).

The multiplication is to be done using more precision (with at least 16-bit dynamics). The sign extension property of the shift operation does not matter here: only the lowest 8 bits are used from the result, and there the sign extension shifting and unsigned shifting are consistent with each other.

Now we describe the contents of color transform data so that decoding can apply the inverse color transform and recover the original red and blue values. The first 3 bits of the color transform data contain the width and height of the image block in number of bits, just like the predictor transform:

```
int size_bits = ReadBits(3) + 2;
int block_width = 1 << size_bits;
int block_height = 1 << size_bits;
```

The remaining part of the color transform data contains ColorTransformElement instances corresponding to each block of the image. ColorTransformElement instances are treated as pixels of an image and encoded using the methods described in Section 7.6.

During decoding, ColorTransformElement instances of the blocks are decoded and the inverse color transform is applied on the ARGB values of the pixels. As mentioned earlier, that inverse color transform is just subtracting ColorTransformElement values from the red and blue channels.

```
void InverseTransform(uint8 red, uint8 green, uint8 blue,
                    ColorTransformElement *p,
                    uint8 *new_red, uint8 *new_blue) {
    // Applying inverse transform is just subtracting the
    // color transform deltas
    red -= ColorTransformDelta(p->green_to_red_, green);
    blue -= ColorTransformDelta(p->green_to_blue_, green);
    blue -= ColorTransformDelta(p->red_to_blue_, red & 0xff);

    *new_red = red & 0xff;
    *new_blue = blue & 0xff;
}
```

7.5.3. Subtract Green Transform

The subtract green transform subtracts green values from red and blue values of each pixel. When this transform is present, the decoder needs to add the green value to both red and blue. There is no data associated with this transform. The decoder applies the inverse transform as follows:

```
void AddGreenToBlueAndRed(uint8 green, uint8 *red, uint8 *blue) {
    *red = (*red + green) & 0xff;
    *blue = (*blue + green) & 0xff;
}
```

This transform is redundant as it can be modeled using the color transform, but it is still often useful. Since it can extend the dynamics of the color transform and there is no additional data here, the subtract green transform can be coded using fewer bits than a full-blown color transform.

7.5.4. Color Indexing Transform

If there are not many unique pixel values, it may be more efficient to create a color index array and replace the pixel values by the array's indices. The color indexing transform achieves this. (In the context of WebP lossless, we specifically do not call this a palette transform because a similar but more dynamic concept exists in WebP lossless encoding: color cache.)

The color indexing transform checks for the number of unique ARGB values in the image. If that number is below a threshold (256), it creates an array of those ARGB values, which is then used to replace the pixel values with the corresponding index: the green channel of the pixels are replaced with the index; all alpha values are set to 255; all red and blue values to 0.

The transform data contains color table size and the entries in the color table. The decoder reads the color indexing transform data as follows:

```
// 8 bit value for color table size
int color_table_size = ReadBits(8) + 1;
```

The color table is stored using the image storage format itself. The color table can be obtained by reading an image, without the RIFF header, image size, and transforms, assuming a height of one pixel and a width of color_table_size. The color table is always subtraction-coded to reduce image entropy. The deltas of palette colors contain typically much less entropy than the colors themselves, leading to significant savings for smaller images. In decoding, every final color in the color table can be obtained by adding the previous color component values by each ARGB component separately, and storing the least significant 8 bits of the result.

The inverse transform for the image is simply replacing the pixel values (which are indices to the color table) with the actual color table values. The indexing is done based on the green component of the ARGB color.

```
// Inverse transform
argb = color_table[GREEN(argb)];
```

If the index is equal or larger than `color_table_size`, the `argb` color value should be set to `0x00000000` (transparent black).

When the color table is small (equal to or less than 16 colors), several pixels are bundled into a single pixel. The pixel bundling packs several (2, 4, or 8) pixels into a single pixel, reducing the image width respectively. Pixel bundling allows for a more efficient joint distribution entropy coding of neighboring pixels, and gives some arithmetic coding-like benefits to the entropy code, but it can only be used when there are a small number of unique values.

`color_table_size` specifies how many pixels are combined together:

```
int width_bits;
if (color_table_size <= 2) {
    width_bits = 3;
} else if (color_table_size <= 4) {
    width_bits = 2;
} else if (color_table_size <= 16) {
    width_bits = 1;
} else {
    width_bits = 0;
}
```

`width_bits` has a value of 0, 1, 2 or 3. A value of 0 indicates no pixel bundling to be done for the image. A value of 1 indicates that two pixels are combined together, and each pixel has a range of `[0..15]`. A value of 2 indicates that four pixels are combined together, and each pixel has a range of `[0..3]`. A value of 3 indicates that eight pixels are combined together and each pixel has a range of `[0..1]`, i.e., a binary value.

The values are packed into the green component as follows:

- * `width_bits = 1`: for every `x` value where $x = 2k + 0$, a green value at `x` is positioned into the 4 least-significant bits of the green value at `x / 2`, a green value at `x + 1` is positioned into the 4 most-significant bits of the green value at `x / 2`.
- * `width_bits = 2`: for every `x` value where $x = 4k + 0$, a green value at `x` is positioned into the 2 least-significant bits of the green value at `x / 4`, green values at `x + 1` to `x + 3` in order to the more significant bits of the green value at `x / 4`.
- * `width_bits = 3`: for every `x` value where $x = 8k + 0$, a green value at `x` is positioned into the least-significant bit of the green value at `x / 8`, green values at `x + 1` to `x + 7` in order to the more significant bits of the green value at `x / 8`.

7.6. Image Data

Image data is an array of pixel values in scan-line order.

7.6.1. Roles of Image Data

We use image data in five different roles:

1. ARGB image: Stores the actual pixels of the image.
2. Entropy image: Stores the meta Huffman codes (Section 7.7.2.1). The red and green components of a pixel define the meta Huffman code used in a particular block of the ARGB image.
3. Predictor image: Stores the metadata for Predictor Transform (Section 7.5.1). The green component of a pixel defines which of the 14 predictors is used within a particular block of the ARGB image.
4. Color transform image. It is created by ColorTransformElement values (defined in Color Transform (Section 7.5.2) for different blocks of the image. Each ColorTransformElement 'cte' is treated as a pixel whose alpha component is 255, red component is cte.red_to_blue, green component is cte.green_to_blue and blue component is cte.green_to_red.
5. Color indexing image: An array of of size color_table_size (up to 256 ARGB values) storing the metadata for the Color Indexing Transform (Section 7.5.4). This is stored as an image of width color_table_size and height 1.

7.6.2. Encoding of Image data

The encoding of image data is independent of its role.

The image is first divided into a set of fixed-size blocks (typically 16x16 blocks). Each of these blocks are modeled using their own entropy codes. Also, several blocks may share the same entropy codes.

Rationale: Storing an entropy code incurs a cost. This cost can be minimized if statistically similar blocks share an entropy code, thereby storing that code only once. For example, an encoder can find similar blocks by clustering them using their statistical properties, or by repeatedly joining a pair of randomly selected clusters when it reduces the overall amount of bits needed to encode the image.

Each pixel is encoded using one of the three possible methods:

1. Huffman coded literal: each channel (green, red, blue and alpha) is entropy-coded independently;
2. LZ77 backward reference: a sequence of pixels are copied from elsewhere in the image; or
3. Color cache code: using a short multiplicative hash code (color cache index) of a recently seen color.

The following sub-sections describe each of these in detail.

7.6.2.1. Huffman Coded Literals

The pixel is stored as Huffman coded values of green, red, blue and alpha (in that order). See Section 7.7.2.2 for details.

7.6.2.2. LZ77 Backward Reference

Backward references are tuples of `_length_` and `_distance code_`:

- * Length indicates how many pixels in scan-line order are to be copied.
- * Distance code is a number indicating the position of a previously seen pixel, from which the pixels are to be copied. The exact mapping is described below (Section 7.6.2.2, Paragraph 11).

The length and distance values are stored using **LZ77 prefix coding**.

LZ77 prefix coding divides large integer values into two parts: the `_prefix code_` and the `_extra bits_`: the prefix code is stored using an entropy code, while the extra bits are stored as they are (without an entropy code).

Rationale: This approach reduces the storage requirement for the entropy code. Also, large values are usually rare, and so extra bits would be used for very few values in the image. Thus, this approach results in a better compression overall.

The following table denotes the prefix codes and extra bits used for storing different range of values.

Note: The maximum backward reference length is limited to 4096. Hence, only the first 24 prefix codes (with the respective extra bits) are meaningful for length values. For distance values, however, all the 40 prefix codes are valid.

Value range	Prefix code	Extra bits
1	0	0
2	1	0
3	2	0
4	3	0
5..6	4	1
7..8	5	1
9..12	6	2
13..16	7	2
...
3072..4096	23	10
...
524289..786432	38	18
786433..1048576	39	18

The pseudocode to obtain a (length or distance) value from the prefix code is as follows:

```

if (prefix_code < 4) {
    return prefix_code + 1;
}
int extra_bits = (prefix_code - 2) >> 1;
int offset = (2 + (prefix_code & 1)) << extra_bits;
return offset + ReadBits(extra_bits) + 1;

```

Distance Mapping:

As noted previously, distance code is a number indicating the position of a previously seen pixel, from which the pixels are to be copied. This sub-section defines the mapping between a distance code and the position of a previous pixel.

The distance codes larger than 120 denote the pixel-distance in scan-line order, offset by 120.

The smallest distance codes [1..120] are special, and are reserved for a close neighborhood of the current pixel. This neighborhood consists of 120 pixels:

- * Pixels that are 1 to 7 rows above the current pixel, and are up to 8 columns to the left or up to 7 columns to the right of the current pixel. [Total such pixels = 7 * (8 + 1 + 7) = 112].
- * Pixels that are in same row as the current pixel, and are up to 8 columns to the left of the current pixel. [8 such pixels].

The mapping between distance code *i* and the neighboring pixel offset (*xi*, *yi*) is as follows:

```
(0, 1), (1, 0), (1, 1), (-1, 1), (0, 2), (2, 0), (1, 2), (-1, 2),
(2, 1), (-2, 1), (2, 2), (-2, 2), (0, 3), (3, 0), (1, 3), (-1, 3),
(3, 1), (-3, 1), (2, 3), (-2, 3), (3, 2), (-3, 2), (0, 4), (4, 0),
(1, 4), (-1, 4), (4, 1), (-4, 1), (3, 3), (-3, 3), (2, 4), (-2, 4),
(4, 2), (-4, 2), (0, 5), (3, 4), (-3, 4), (4, 3), (-4, 3), (5, 0),
(1, 5), (-1, 5), (5, 1), (-5, 1), (2, 5), (-2, 5), (5, 2), (-5, 2),
(4, 4), (-4, 4), (3, 5), (-3, 5), (5, 3), (-5, 3), (0, 6), (6, 0),
(1, 6), (-1, 6), (6, 1), (-6, 1), (2, 6), (-2, 6), (6, 2), (-6, 2),
(4, 5), (-4, 5), (5, 4), (-5, 4), (3, 6), (-3, 6), (6, 3), (-6, 3),
(0, 7), (7, 0), (1, 7), (-1, 7), (5, 5), (-5, 5), (7, 1), (-7, 1),
(4, 6), (-4, 6), (6, 4), (-6, 4), (2, 7), (-2, 7), (7, 2), (-7, 2),
(3, 7), (-3, 7), (7, 3), (-7, 3), (5, 6), (-5, 6), (6, 5), (-6, 5),
(8, 0), (4, 7), (-4, 7), (7, 4), (-7, 4), (8, 1), (8, 2), (6, 6),
(-6, 6), (8, 3), (5, 7), (-5, 7), (7, 5), (-7, 5), (8, 4), (6, 7),
(-6, 7), (7, 6), (-7, 6), (8, 5), (7, 7), (-7, 7), (8, 6), (8, 7)
```

For example, distance code 1 indicates offset of (0, 1) for the neighboring pixel, that is, the pixel above the current pixel (0-pixel difference in X-direction and 1 pixel difference in Y-direction). Similarly, distance code 3 indicates left-top pixel.

The decoder can convert a distances code '*i*' to a scan-line order distance '*dist*' as follows:

```
(xi, yi) = distance_map[i]
dist = x + y * xsize
if (dist < 1) {
    dist = 1
}
```

where '*distance_map*' is the mapping noted above and *xsize* is the width of the image in pixels.

7.6.2.3. Color Cache Coding

Color cache stores a set of colors that have been recently used in the image.

**Rationale:* This way, the recently used colors can sometimes be referred to more efficiently than emitting them using other two methods (described in Section 7.6.2.1 and Section 7.6.2.2).

Color cache codes are stored as follows. First, there is a 1-bit value that indicates if the color cache is used. If this bit is 0, no color cache codes exist, and they are not transmitted in the Huffman code that decodes the green symbols and the length prefix codes. However, if this bit is 1, the color cache size is read next:

```
int color_cache_code_bits = ReadBits(4);
int color_cache_size = 1 << color_cache_code_bits;
```

color_cache_code_bits defines the size of the color_cache by $(1 \ll \text{color_cache_code_bits})$. The range of allowed values for color_cache_code_bits is [1..11]. Compliant decoders must indicate a corrupted bitstream for other values.

A color cache is an array of size color_cache_size. Each entry stores one ARGB color. Colors are looked up by indexing them by $(0x1e35a7bd * \text{color}) \gg (32 - \text{color_cache_code_bits})$. Only one lookup is done in a color cache; there is no conflict resolution.

In the beginning of decoding or encoding of an image, all entries in all color cache values are set to zero. The color cache code is converted to this color at decoding time. The state of the color cache is maintained by inserting every pixel, be it produced by backward referencing or as literals, into the cache in the order they appear in the stream.

7.7. Entropy Code

7.7.1. Overview

Most of the data is coded using canonical Huffman code [huffman]. Hence, the codes are transmitted by sending the _Huffman code lengths_, as opposed to the actual _Huffman codes_.

In particular, the format uses **spatially-variant Huffman coding**. In other words, different blocks of the image can potentially use different entropy codes.

Rationale: Different areas of the image may have different characteristics. So, allowing them to use different entropy codes provides more flexibility and potentially a better compression.

7.7.2. Details

The encoded image data consists of two parts:

1. Meta Huffman codes

2. Entropy-coded image data

7.7.2.1. Decoding of Meta Huffman Codes

As noted earlier, the format allows the use of different Huffman codes for different blocks of the image. `_Meta Huffman codes_` are indexes identifying which Huffman codes to use in different parts of the image.

Meta Huffman codes may be used `_only_` when the image is being used in the role (Section 7.6.1) of an `_ARGB image_`.

There are two possibilities for the meta Huffman codes, indicated by a 1-bit value:

- * If this bit is zero, there is only one meta Huffman code used everywhere in the image. No more data is stored.
- * If this bit is one, the image uses multiple meta Huffman codes. These meta Huffman codes are stored as an `_entropy image_` (described below).

`*Entropy image:*`

The entropy image defines which Huffman codes are used in different parts of the image, as described below.

The first 3-bits contain the `huffman_bits` value. The dimensions of the entropy image are derived from `'huffman_bits'`.

```
int huffman_bits = ReadBits(3) + 2;
int huffman_xsize = DIV_ROUND_UP(xsize, 1 << huffman_bits);
int huffman_ysize = DIV_ROUND_UP(ysize, 1 << huffman_bits);
```

where `DIV_ROUND_UP` is as defined in Section 7.5.1.

Next bits contain an entropy image of width `huffman_xsize` and height `huffman_ysize`.

`*Interpretation of Meta Huffman Codes:*`

For any given pixel `(x, y)`, there is a set of five Huffman codes associated with it. These codes are (in bitstream order):

- * `*Huffman code #1*`: used for green channel, backward-reference length and color cache

* *Huffman code #2, #3 and #4*: used for red, blue and alpha channels respectively.

* *Huffman code #5*: used for backward-reference distance.

From here on, we refer to this set as a *Huffman code group*.

The number of Huffman code groups in the ARGB image can be obtained by finding the `_largest meta Huffman code_` from the entropy image:

```
int num_huff_groups = max(entropy image) + 1;
```

where `max(entropy image)` indicates the largest Huffman code stored in the entropy image.

As each Huffman code groups contains five Huffman codes, the total number of Huffman codes is:

```
int num_huff_codes = 5 * num_huff_groups;
```

Given a pixel `(x, y)` in the ARGB image, we can obtain the corresponding Huffman codes to be used as follows:

```
int position = (y >> huffman_bits) * huffman_xsize + (x >> huffman_bits);
int meta_huff_code = (entropy_image[pos] >> 8) & 0xffff;
HuffmanCodeGroup huff_group = huffman_code_groups[meta_huff_code];
```

where, we have assumed the existence of `HuffmanCodeGroup` structure, which represents a set of five Huffman codes. Also, `huffman_code_groups` is an array of `HuffmanCodeGroup` (of size `num_huff_groups`).

The decoder then uses Huffman code group `huff_group` to decode the pixel `(x, y)` as explained in Section 7.7.2.2.

7.7.2.2. Decoding Entropy-coded Image Data

For the current position `(x, y)` in the image, the decoder first identifies the corresponding Huffman code group (as explained in the last section). Given the Huffman code group, the pixel is read and decoded as follows:

Read next symbol `S` from the bitstream using Huffman code #1. [See Section 7.7.2.2, Paragraph 5 for details on decoding the Huffman code lengths]. Note that `S` is any integer in the range 0 to $(256 + 24 + \text{color_cache_size}) - 1$ (Section 7.6.2.3) - 1).

The interpretation of `S` depends on its value:

1. if $S < 256$
 - i. Use S as the green component
 - ii. Read red from the bitstream using Huffman code #2
 - iii. Read blue from the bitstream using Huffman code #3
 - iv. Read alpha from the bitstream using Huffman code #4
2. if $S < 256 + 24$
 - i. Use $S - 256$ as a length prefix code
 - ii. Read extra bits for length from the bitstream
 - iii. Determine backward-reference length L from length prefix code and the extra bits read.
 - iv. Read distance prefix code from the bitstream using Huffman code #5
 - v. Read extra bits for distance from the bitstream
 - vi. Determine backward-reference distance D from distance prefix code and the extra bits read.
 - vii. Copy the L pixels (in scan-line order) from the sequence of pixels prior to them by D pixels.
3. if $S \geq 256 + 24$
 - i. Use $S - (256 + 24)$ as the index into the color cache.
 - ii. Get ARGB color from the color cache at that index.

Decoding the Code Lengths:

This section describes the details about reading a symbol from the bitstream by decoding the Huffman code length.

The Huffman code lengths can be coded in two ways. The method used is specified by a 1-bit value.

* If this bit is 1, it is a `_simple code length code_`, and

* If this bit is 0, it is a `_normal code length code_`.

(i) Simple Code Length Code:

This variant is used in the special case when only 1 or 2 Huffman code lengths are non-zero, and are in the range of [0, 255]. All other Huffman code lengths are implicitly zeros.

The first bit indicates the number of non-zero code lengths:

```
int num_code_lengths = ReadBits(1) + 1;
```

The first code length is stored either using a 1-bit code for values of 0 and 1, or using an 8-bit code for values in range [0, 255]. The second code length, when present, is coded as an 8-bit code.

```
int is_first_8bits = ReadBits(1);
code_lengths[0] = ReadBits(1 + 7 * is_first_8bits);
if (num_code_lengths == 2) {
    code_lengths[1] = ReadBits(8);
}
```

Note: Another special case is when all Huffman code lengths are zeros (an empty Huffman code). For example, a Huffman code for distance can be empty if there are no backward references. Similarly, Huffman codes for alpha, red, and blue can be empty if all pixels within the same meta Huffman code are produced using the color cache. However, this case doesn't need a special handling, as empty Huffman codes can be coded as those containing a single symbol 0.

(ii) Normal Code Length Code:

The code lengths of a Huffman code are read as follows: `num_code_lengths` specifies the number of code lengths; the rest of the code lengths (according to the order in `kCodeLengthCodeOrder`) are zeros.

```
int kCodeLengthCodes = 19;
int kCodeLengthCodeOrder[kCodeLengthCodes] = {
    17, 18, 0, 1, 2, 3, 4, 5, 16, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
};
int code_lengths[kCodeLengthCodes] = { 0 }; // All zeros.
int num_code_lengths = 4 + ReadBits(4);
for (i = 0; i < num_code_lengths; ++i) {
    code_lengths[kCodeLengthCodeOrder[i]] = ReadBits(3);
}
```

* Code length code [0..15] indicates literal code lengths.

- Value 0 means no symbols have been coded.

- Values [1..15] indicate the bit length of the respective code.
- * Code 16 repeats the previous non-zero value [3..6] times, i.e., 3 + ReadBits(2) times. If code 16 is used before a non-zero value has been emitted, a value of 8 is repeated.
- * Code 17 emits a streak of zeros [3..10], i.e., 3 + ReadBits(3) times.
- * Code 18 emits a streak of zeros of length [11..138], i.e., 11 + ReadBits(7) times.

7.8. Overall Structure of the Format

Below is a view into the format in Backus-Naur form. It does not cover all details. End-of-image (EOI) is only implicitly coded into the number of pixels (xsize * ysize).

7.8.1. Basic Structure

```
<format> ::= <RIFF header><image size><image stream>
<image stream> ::= <optional-transform><spatially-coded image>
```

7.8.2. Structure of Transforms

```
<optional-transform> ::= (1-bit value 1; <transform> <optional-transform>) |
                          1-bit value 0
<transform> ::= <predictor-tx> | <color-tx> | <subtract-green-tx> |
                <color-indexing-tx>
<predictor-tx> ::= 2-bit value 0; <predictor image>
<predictor image> ::= 3-bit sub-pixel code ; <entropy-coded image>
<color-tx> ::= 2-bit value 1; <color image>
<color image> ::= 3-bit sub-pixel code ; <entropy-coded image>
<subtract-green-tx> ::= 2-bit value 2
<color-indexing-tx> ::= 2-bit value 3; <color-indexing image>
<color-indexing image> ::= 8-bit color count; <entropy-coded image>
```

7.8.3. Structure of the Image Data

```

<spatially-coded image> ::= <meta huffman><entropy-coded image>
<entropy-coded image> ::= <color cache info><huffman codes><lz77-coded image>
<meta huffman> ::= 1-bit value 0 |
                    (1-bit value 1; <entropy image>)
<entropy image> ::= 3-bit subsample value; <entropy-coded image>
<color cache info> ::= 1 bit value 0 |
                    (1-bit value 1; 4-bit value for color cache size)
<huffman codes> ::= <huffman code group> | <huffman code group><huffman codes>
<huffman code group> ::= <huffman code><huffman code><huffman code>
                        <huffman code><huffman code>
                        See "Interpretation of Meta Huffman codes" to
                        understand what each of these five Huffman codes are
                        for.
<huffman code> ::= <simple huffman code> | <normal huffman code>
<simple huffman code> ::= see "Simple code length code" for details
<normal huffman code> ::= <code length code>; encoded code lengths
<code length code> ::= see section "Normal code length code"
<lz77-coded image> ::= ((<argb-pixel> | <lz77-copy> | <color-cache-code>)
                        <lz77-coded image>) | ""

```

A possible example sequence:

```

<RIFF header><image size>1-bit value 1<subtract-green-tx>
1-bit value 1<predictor-tx>1-bit value 0<meta huffman>
<color cache info><huffman codes>
<lz77-coded image>

```

8. References

8.1. Normative References

- [rec601] ITU, "BT.601: Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios", March 2011, <<https://www.itu.int/rec/R-REC-BT.601/>>.
- [RFC1166] Kirkpatrick, S., Stahl, M., and M. Recker, "Internet numbers", RFC 1166, DOI 10.17487/RFC1166, July 1990, <<https://www.rfc-editor.org/info/rfc1166>>.
- [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>>.
- [RFC2781] Hoffman, P. and F. Yergeau, "UTF-16, an encoding of ISO 10646", RFC 2781, DOI 10.17487/RFC2781, February 2000, <<https://www.rfc-editor.org/info/rfc2781>>.

- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC6386] Bankoski, J., Koleszar, J., Quillio, L., Salonen, J., Wilkins, P., and Y. Xu, "VP8 Data Format and Decoding Guide", RFC 6386, DOI 10.17487/RFC6386, November 2011, <<https://www.rfc-editor.org/info/rfc6386>>.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <<https://www.rfc-editor.org/info/rfc6838>>.
- [webp-lossless-src] Alakuijala, J., "WebP Lossless Bitstream Specification", September 2014, <<https://chromium.googlesource.com/webm/libwebp/+/refs/heads/main/doc/webp-lossless-bitstream-spec.txt>>.
- [webp-riff-src] Google LLC, "WebP RIFF Container", April 2018, <<https://chromium.googlesource.com/webm/libwebp/+/refs/heads/main/doc/webp-container-spec.txt>>.

8.2. Informative References

- [crbug-security] "libwebp Security Issues", <<https://bugs.chromium.org/p/webp/issues/list?q=label%3ASecurity>>.
- [cve.mitre.org-libwebp] "libwebp CVE List", <<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=libwebp>>.
- [Exif] Camera & Imaging Products Association (CIPA), Japan Electronics and Information Technology Industries Association (JEITA), "Exchangeable image file format for digital still cameras: Exif Version 2.3", <https://www.cipa.jp/std/documents/e/DC-008-2012_E.pdf>.
- [gif-spec] "GIF89a Specification", <<https://www.w3.org/Graphics/GIF/spec-gif89a.txt>>.
- [huffman] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes", Proceedings of the Institute of Radio Engineers Number 9, pp. 1098-1101., September 1952.

- ```
[IANA-Media-Types] Internet Assigned Numbers Authority (IANA), "Media Types",
 <https://www.iana.org/assignments/media-types/media-
 types.xhtml>.

[ICC] International Color Consortium, "ICC Specification",
 December 2010,
 <https://www.color.org/specification/ICC1v43_2010-12.pdf>.

[jpeg-spec] "JPEG Standard (JPEG ISO/IEC 10918-1 ITU-T Recommendation
 T.81)", <https://www.w3.org/Graphics/JPEG/itu-t81.pdf>.

[lz77] Ziv, J. and A. Lempel, "A Universal Algorithm for
 Sequential Data Compression", IEEE Transactions on
 Information Theory Vol. 23, No. 3, pp. 337-343., May 1977.

[mwg] Metadata Working Group, "Guidelines For Handling Image
 Metadata", November 2010,
 <https://web.archive.org/web/20180919181934/
 http://www.metadataworkinggroup.org/pdf/mwg_guidance.pdf>.

[RFC2083] Boutell, T., "PNG (Portable Network Graphics)
 Specification Version 1.0", RFC 2083,
 DOI 10.17487/RFC2083, March 1997,
 <https://www.rfc-editor.org/info/rfc2083>.

[riff-spec] "Multimedia Programming Interface and Data Specifications
 1.0", <http://www-
 mmstp.ece.mcgill.ca/Documents/AudioFormats/WAVE/Docs/
 riffmci.pdf>.

[XMP] Adobe Inc., "XMP Specification",
 <https://www.adobe.com/devnet/xmp.html>.
```

## Author's Address

James Zern  
Google LLC  
1600 Amphitheatre Parkway  
Mountain View, CA 94043  
United States of America

Phone: +1 650 253-0000  
Email: jzern@google.com