mlcodec
Internet-Draft
Updates: 6716 (if approved)
Intended status: Standards Track
Expires: 9 May 2026

T. Terriberry
Xiph.Org
JM. Valin
Google

5 November 2025

Extension Formatting for the Opus Codec
draft-ietf-mlcodec-opus-extension-05

Abstract

This document updates RFC6716 to extend the Opus codec (RFC6716) in a way that maintains interoperability, while adding optional functionality.

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 9 May 2026.

Copyright Notice

Table of Contents

1. Introduction 2
1.1. Requirements Language 3
2. Extension Format 3
2.1. ID 0: Original Padding 5
2.2. ID 1: Frame Separator 5
2.3. ID 2: Repeat These Extensions (RTE) 6
2.4. IDs 3-119: Unassigned 7
2.5. IDs 120-126: Experimental 7
2.6. ID 127: Extended Extensions 8
2.7. Discard When Out-of-Bounds 8
3. IANA Considerations 8
3.1. Opus Media Type Update 9
3.2. Mapping to SDP Parameters 10
4. Security Considerations 10
5. References 11
5.1. Normative References 11
Appendix A. Examples 11
A.1. Example 1 12
A.2. Example 2 13
A.3. Example 3 13
A.4. Example 4 13
A.5. Example 5 13
A.6. Example 5 14
A.7. Example 6 14
A.8. Example 7 14
A.9. Example 8 15
A.10. Example 9 15
A.11. Example 10 16
A.12. Example 11 16
A.13. Example 12 17
Authors' Addresses 17

1. Introduction

This document updates RFC6716 to extend the Opus codec (RFC6716) in a way that maintains interoperability, while adding optional functionality. Pre-existing decoders that comply with RFC6716 will safely ignore these extensions. Extended decoders that comply with this standard MUST behave identically to a non-extended decoder when extensions are absent, ensuring backwards compatibility.

1.1. Requirements Language

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. Extension Format

The Opus padding mechanism provides a safe way to extend the Opus codec while preserving interoperability and without having to transmit any extra packets. [RFC6716] specifies that all padding bytes "MUST be set to zero" by the encoder, while the decoder "MUST accept any value for the padding bytes". In that way, any non-zero padding will indicate to an extended decoder that extensions are present and can be processed. On the other hand, for any all-zero padding, the decoder will just discard the padding like any non-extended decoder. A non-extended decoder receiving a packet with extensions will simply discard the extensions and proceed as if none were present.

An instance of an extension is composed of an "extension ID byte" and an optional payload, which may be prefixed by an optional length indicator, followed by 0 or more bytes of extension data. Although there is only one padding region per Opus packet, each extension instance is tied to an underlying Opus frame, of which there may be more than one per packet. Extension instances are grouped by the corresponding Opus frame they are extending, starting from the first frame, with frame separator extensions (Section 2.2) delineating the boundaries between the extensions for each frame.

There are three types of extensions:

* Structural extensions (IDs 0, 1, and 2), which control extension parsing, but do not inherently change the behavior of a decoder themselves.

* Short extensions (IDs 3 through 31), which have either 0 or 1 bytes of extension data.

* Long extensions (IDs 32 through 127), which can have an arbitrary number of extension data bytes.

An extension instance starts with an "extension ID byte" that contains a 7-bit ID, as well as a binary flag L for length signaling. L is the least significant bit of the extension ID byte, with the upper 7 bits encoding the ID. For short extensions, L=0 means that

no data follows the extension ID byte, whereas L=1 means that exactly one byte of extension data follows. For long extensions, L=0 signals that the extension data takes up the rest of the padding. In any given packet, this signal cannot appear more than once. Conversely, L=1 in a long extension signals that a length indicator follows. The following byte contains a length value from 0 to 254, or the special value 255, indicating that the length is 255 plus the length signaled from the next byte. The 255 case MAY repeat as long as the size of the padding is not exceeded. Any possible buffer overflow is handled by ignoring the offending extension (see Section 2.7).

For ID 0 (Original Padding), L=0 has the same meaning as for long extensions, but L=1 signals a length of zero (no length indicator or extension data follows). For ID 1 (Frame Separator), the L flag has the same meaning as for short extensions. For ID 2 (Repeat These Extensions), the extension itself has no payload (for either L=0 or L=1), but is used to signal that previously coded extensions are to be repeated for subsequent Opus frames. The payloads of the repeated extensions follow immediately after. See Section 2.3 for the details of this process and for how the L flag is to be interpreted.

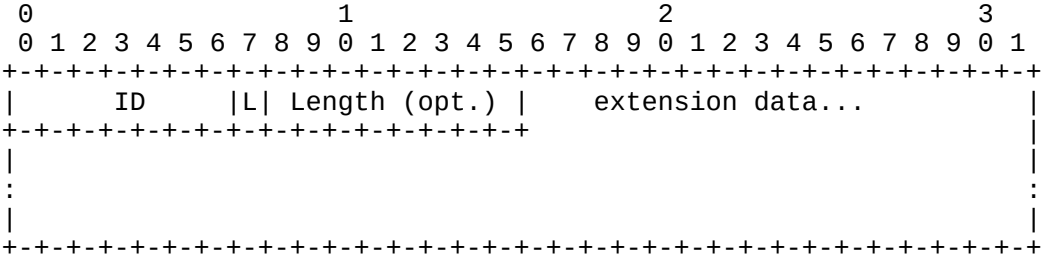

Figure 1: Extension framing

A decoder MUST ignore any extension it does not support, decoding the rest of the packet as if the extension was not present. Additionally, a decoder MAY ignore any other extension even if it technically supports it. An encoder MUST NOT alter the way it encodes the non-extension part of an Opus packet in such a way as to noticeably reduce its quality when decoded with a non-extended decoder.

A given extension ID MAY appear multiple times. The ordering of extension instances within each Opus frame is significant (see Section 2.2). A particular extension ID definition MAY place further restrictions on the count and the ordering of these extensions instances (see Section 3). Reordering of extension instances between Opus frames caused by the repeat mechanism is not significant, and an extended decoder MUST treat repeated extensions as equivalent to the same extensions coded individually (see Section 2.3).

2.1. ID 0: Original Padding

For compatibility reasons, an ID of 0 means that the remaining content of the padding is actual padding, as originally defined in [RFC6716]. As in its original definition, the padding bytes MUST be set to zero by the encoder, while the decoder MUST ignore any non-zero padding. In the case where the L flag is set, the extension ID byte (0x01) is simply skipped, and extension decoding continues from the next byte. This allows inserting padding one byte at a time in a way that would not be possible if an explicit padding length were coded instead (that would make L=1 padding require at least two bytes, and appending a L=0 padding might require signaling a multi-byte length indicator for a preceding long extension that would not otherwise be necessary, both causing the packet size to increase by more than one byte).

2.2. ID 1: Frame Separator

In the case where multiple Opus frames are packed inside the same packet, frame separators specify which extension instance(s) are associated with which frames. An extension instance with ID=1 acts as a separator between extension instances from different Opus frames.

By default, extension instances are associated with the first Opus frame in the packet (frame 0). When parsing sequentially, any time a separator with L=0 is encountered, the associated frame index is incremented by one. If L=1 is used, the following payload byte indicates the amount by which to increment the frame index. The frame index MUST NOT exceed the number of frames in the packet minus one (i.e., indexing starts at zero), regardless of how it is incremented. The decoder MUST ignore all extension instances associated with an out-of-bounds frame index (see Section 2.7).

2.3. ID 2: Repeat These Extensions (RTE)

In the case where multiple Opus frames are packed inside the same packet, the Repeat These Extensions (RTE) extension can reduce the overhead of coding extension IDs and frame separators when the extensions in the current frame also appear in every subsequent frame (albeit, possibly with different payloads). An extension with ID=2 acts as a signal to repeat all of the non-padding (ID=0) extensions following the most recent of

* The start of the packet, or

* A frame separator (ID=1) with a non-zero increment, or

* A preceding RTE extension (ID=2), if any.

Padding extensions are not repeated, nor is any frame separator with an increment of 0 (which acts as another form of padding). Extensions preceding a frame separator with an increment of zero do get repeated, as they still belong to the current frame. An RTE extension MAY appear multiple times in the same frame. Only the extensions that follow the most recent RTE (if any) are repeated by a subsequent RTE.

The RTE extension itself has no payload, but it is immediately followed by the payloads of new instances of the repeated extensions. The payloads for all of the repeated extensions for the next frame come first, followed by those of the frame after, etc. The extension ID byte corresponding to each payload is implicit and not coded: only the length (if needed) and the subsequent extension data are coded. An RTE extension MAY appear in the last frame. In this case, no extensions are repeated.

For short extensions, the repeated extension payloads use the same L flag as the instance of the extension being repeated. If the length of a short extension needs to change between frames, this repeat mechanism cannot be used to signal that. All repeated long extension payloads except the final instance of the last repeated long extension in the last frame are coded as if with L=1 (using an explicit length indicator). The final repeated long extension payload is coded with the L flag specified by the RTE extension. In the case that the RTE extension specifies L=0, and the last repeated long extension is followed by one or more repeated short extensions with a payload, then the final long extension does not consume the rest of the padding as normal, but leaves enough room for the payloads of the repeated short extensions that follow. If there is not enough room for the repeated short extensions that follow, even if the length of the final long extension were set to zero, then that extension instance and all remaining padding data MUST be ignored by the decoder (see Section 2.7).

If the RTE extension uses L=1, then extension coding continues afterwards with the same frame index as the RTE extension. This allows a frame to contain both repeated and non-repeated extensions. This also means that the complete collection of extension instances for a given frame might not all be contiguous in the packet. If the RTE extension uses L=0, but the repeated extensions did not contain a long extension, then extension coding continues afterwards with the frame index following that of the RTE extension (as if an L=0 separator had been coded). If an RTE extension with L=0 appears in the last frame, then the rest of the padding (if any) MUST be set to zero by the encoder, and the decoder MUST ignore any additional non-zero padding.

2.4. IDs 3-119: Unassigned

These extensions are to be defined in their own respective documents, and the IDs are to be assigned by IANA. The meaning of the L flag is already defined for all of these unassigned IDs because a decoder must know how to skip extensions it does not support. Due to the potential for interaction between extensions, new extensions are to be assigned with the "Standards Action" policy defined by [RFC8126].

2.5. IDs 120-126: Experimental

We reserve these 7 IDs for experimental extensions, such that extensions defined in Internet-Drafts can be tested before publication as an RFC without causing possible interoperability issues should their bitstream definitions change. When using an experimental ID, it is RECOMMENDED to use a two-byte prefix that attempts to encode an experiment number (first byte) and a version

number (second byte). Experimental extension documents SHOULD attempt to choose an experiment number that does not collide with other ongoing experiments.

2.6. ID 127: Extended Extensions

The last ID is reserved for future extensions to the extension mechanism. As with all other long extensions, the meaning of the L flag is pre-defined to ensure decoders can skip extended extensions they do not support. The contents of the payload for this extension will be defined by a future specification.

2.7. Discard When Out-of-Bounds

Any extension signaled with a length that would cause the decoder to read beyond the bounds of the packet MUST be ignored by the decoder. This includes short extensions as well as long, and for long extensions also includes the case where the length itself cannot be fully decoded without reading beyond the bounds of the packet. It also includes the case where the packet does not contain enough padding for the payloads of extensions repeated by the RTE extension (ID=2).

Similarly, any extension signaled with a frame index greater than or equal to the number of Opus frames in the packet MUST be ignored by the decoder.

3. IANA Considerations

This document defines a new registry, "Opus Extension IDs," in a new "Opus" group, that allocates individual IDs to individual extensions to be defined in the future.

Registry Name: Opus Extension IDs

Group: Opus

Value: 0...127 (7-bit)

Registration Policy: "Standard Action"

Registration template MUST include: ID, description, any restrictions on multiplicity or ordering, extension interaction notes, any media type parameters, and security considerations reference

The existing "Opus Channel Mapping Families" registry will also be moved to the newly created "Opus" group. Moreover, this document already defines the following IDs:

Extension ID	Description	Reference
0	Original Padding	Defined in Section 2.1
1	Frame Separator	Defined in Section 2.2.
2	Repeat These Extensions	Defined in Section 2.3.
3-119	Unassigned	To be assigned with the "Standards Action" policy [RFC8126]
120-126	Experimental	Defined in Section 2.5, following the "Experimental Use" policy [RFC8126]
127	Extended Extensions	Reserved in Section 2.6

Table 1

For forward compatibility, any extension MUST use the definition of the L flag dictated by its ID value (see Section 2). Extension definitions MUST specify whether or not it is permitted for the extension to appear multiple times for a given Opus frame within the packet.

3.1. Opus Media Type Update

This document updates the audio/opus media type registration [RFC7587] to add the following two optional parameters:

extensions: specifies a comma-separated list of supported extension IDs on the receiver side.

sprop-extensions: specifies a comma-separated list of supported extension IDs on the sender side.

extN-*: To facilitate parameter forwarding, extension document that require receiver extension parameters SHOULD name them "ext", followed by the extension ID, a hyphen, and the parameter name.

sprop-extN-*: Extension-specific sender-side parameters defined similarly as above.

All names starting with "ext" and "sprop-ext" are reserved for use by Opus extensions. Unknown extN-* and sprop-extN-* parameters MUST be ignored unless extension ID N is included in the corresponding list.

Structural extensions (IDs 0, 1, and 2) MUST be supported by any receiver that recognizes Opus extensions. They do not need to be included in the extensions or sprop-extensions lists.

The following is the formal Augmented Backus-Naur Form (ABNF) [RFC5234] syntax specifying the parameters described above.

```
; DIGIT defined in RFC5234
EXTENSION-ID =  %x31-39 *2DIGIT
EXTENSION-LIST = EXTENSION-ID *("," EXTENSION-ID)
extensions = EXTENSION-LIST
sprop-extensions = EXTENSION-LIST
; ALPHA defined in RFC5234
ext-param-name = "ext"  EXTENSION-ID "-" 1*114(ALPHA / DIGIT / "-")
sprop-ext-param-name = "sprop-ext" EXTENSION-ID "-"
                       1*114(ALPHA / DIGIT / "-")
```

Figure 2: Media Type Parameter ABNF

3.2. Mapping to SDP Parameters

The media type parameters described above map to declarative SDP and SDP offer-answer in the same way as other optional parameters in [RFC7587]. The media-level format parameters described in this document MUST be explicitly specified and MUST NOT be carried over blindly from another offer or answer. Regardless of any a=fmtp SDP attribute specified, the receiver MUST be capable of decoding any unknown or non-negotiated extensions, even if the negotiation fails.

4. Security Considerations

This document does not add security considerations beyond those already documented in [RFC6716]. We repeat for emphasis that implementations of these extension mechanisms need to be robust against malicious payloads. Malicious payloads must not cause a decoder to overrun its allocated memory or to take an excessive amount of resources to decode. Future Opus extensions may have their

own security implications.

5. References

5.1. 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>.

[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>.

[RFC6716] Valin, JM., Vos, K., and T. Terriberry, "Definition of the Opus Audio Codec", RFC 6716, DOI 10.17487/RFC6716, September 2012, <https://www.rfc-editor.org/info/rfc6716>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC7587] Spittka, J., Vos, K., and JM. Valin, "RTP Payload Format for the Opus Speech and Audio Codec", RFC 7587, DOI 10.17487/RFC7587, June 2015, <https://www.rfc-editor.org/info/rfc7587>.

[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>.

Appendix A. Examples

This appendix give some examples of extensions, their payloads, and the corresponding encoded bytes stored in the padding of an Opus packet. Both long and short extension IDs are used to illustrate their encodings, but no experimental range is defined for short extensions. The examples here are examples only, and should not be taken as valid encodings of real extensions using those IDs, which may yet be defined elsewhere.

All of the examples here are for a 60ms Opus packet with 3 x 20ms Opus frames. They are created by encoding different subsets of the following list of extension IDs and payloads:

Index	ID	Frame	Data	(Length)
0	28	0	"a"	1
1	28	1	"b"	1
2	28	2	"c"	1
3	29	0	"d"	1
4	29	1		0
5	29	2		0
6	120	2	"E0ex2"	5
7	120	1	"E0ex"	4
8	30	1		0
9	30	2		0
10	31	2	"f"	1
11	31	1	"e"	1
12	120	1	"E0example"	9

Table 2

For ease of constructing the examples, the entries in the table are not ordered by frame. When validating the results of parsing any of these examples, the order of the extensions within a frame should match their order in this table, even if they are not contiguous in the table.

A.1. Example 1

{0x39, 0x61}

Figure 3: Encoding of index 0 (2 bytes)

This illustrates the basic encoding of a short extension with a 1-byte payload into two octets. The extension ID (28) is combined with the L-flag (1) into the decimal value 2*28 + 1 = 57 (hex 0x39). The payload byte is encoded as-is.

A.2. Example 2

{0x39, 0x61, 0x02, 0x39, 0x62}

Figure 4: Encoding of indices 0...1 (5 bytes)

This illustrates the encoding of two extensions in different frames. A frame separator (ID=1, L=0, hex encoding 0x02) separates the extensions for each frame. Even though these extensions share the same ID, the repeat mechanism cannot be used, because the packet has three frames, and the extension is not present in the last frame.

A.3. Example 3

{0x39, 0x61, 0x04, 0x62, 0x63}

Figure 5: Encoding of indices 0...2 (5 bytes)

Once the extension is also included in the last frame, the repeat mechanism can be used (ID=2, L=0, hex encoding 0x04). This encoding has the same length as the previous example, even though we added an extra byte of payload, thanks to the savings from not having to signal the extension ID repeatedly or the frame separators.

A.4. Example 4

{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64}

Figure 6: Encoding of indices 0...3 (7 bytes)

This shows both a repeated extension and a non-repeated extension in the same packet. The payloads for the repeated extension in subsequent frames are encoded immediately, and only afterwards is the non-repeated extension encoded. The repeat indicator (ID=2) now needs the L flag set to 1 (hex encoding 0x05), to indicate that may be additional extensions in the current (first) frame. No frame separators are required here.

A.5. Example 5

{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02, 0x3A}

Figure 7: Encoding of indices 0...4 (9 bytes)

Adding another instance of the second extension to the next, we still cannot use the repeat mechanism with it, and now do need a frame separator.

A.6. Example 5

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02, 0x3A, 0x04}
```

Figure 8: Encoding of indices 0...5 (10 bytes)

Now we do have the same two extensions in all three frames, but initially we cannot repeat the ID=29 extensions, because they are short extensions and the L flags do not match in all the frames. After the first frame, the L flags do match for the remaining frames, and we can use a repeat to save having to code another frame separator. The following encoding is also valid, if inefficient:

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02, 0x3A, 0x04, 0x04, 0x00}
```

Figure 9: Encoding of indices 0...5 (12 bytes)

Here an additional repeat has been signaled in the last frame, but there are no new extensions to repeat (also there are no subsequent frames to repeat them in). The extra trailing 0x00 byte, and any additional bytes, are ignored.

A.7. Example 6

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0x04, 0xF0, 0x45, 0x30, 0x65, 0x78, 0x32}
```

Figure 10: Encoding of indices 0...6 (16 bytes)

Now we add a long extension. Because the only extensions in subsequent frames have already been signaled using the repeat mechanism, it can be encoded with L=0 to indicate its payload takes the rest of the packet and to avoid signaling a length.

A.8. Example 7

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0xF1, 0x04, 0x45, 0x30, 0x65, 0x78, 0x04,
 0x45, 0x30, 0x65, 0x78, 0x32}
```

Figure 11: Encoding of indices 0...7 (21 bytes)

By adding a long extension with the same ID to the last frame, we can now use the repeat mechanism to repeat multiple extensions at a time. The first instance of the long extension requires a length (hex encoding 0x04). Then an L=0 repeat is signaled (hex encoding also 0x04). The final instance of the long extension does not require a length: its payload consume the rest of the packet. The following encoding is also valid:

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0xF1, 0x04, 0x45, 0x30, 0x65, 0x78, 0x01,
 0x04, 0x45, 0x30, 0x65, 0x78, 0x32}
```

Figure 12: Encoding of indices 0...7 (22 bytes)

Here a padding byte (ID=0, L=1, hex encoding 0x01) has been inserted before the second repeat. Since it is padding, it does not get repeated. Additionally, this encoding is also valid:

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x05, 0x3B, 0x64,
 0x02, 0x3A, 0x05, 0xF1, 0x04, 0x45, 0x30, 0x65,
 0x78, 0x01, 0x04, 0x45, 0x30, 0x65, 0x78, 0x32}
```

Figure 13: Encoding of indices 0...7 (24 bytes)

Here we have inserted two extra repeat indicators (ID=2, L=1, hex encoding=0x05). In the first case, there are no new extensions to repeat, so this simply acts as another form of padding. In the second case, we now have multiple repeats in the same frame. If the short extensions being repeated (ID=29, L=0) had payloads, they would now all come before the payloads for the long extensions (ID=120), instead of being interleaved.

A.9. Example 8

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0xF1, 0x04, 0x45, 0x30, 0x65, 0x78, 0x05,
 0x05, 0x45, 0x30, 0x65, 0x78, 0x32, 0x3C}
```

Figure 14: Encoding of indices 0...8 (23 bytes)

Now we add another short extension to the second frame. This means we can no longer skip coding the length for the final long extension.

A.10. Example 9

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0xF1, 0x04, 0x45, 0x30, 0x65, 0x78, 0x3C,
 0x04, 0x45, 0x30, 0x65, 0x78, 0x32}
```

Figure 15: Encoding of indices 0...9 (22 bytes)

However, if the same short extension is also included in the third frame, we can repeat them both. Now we are once again able to avoid coding the length for the final long extension, even though it is followed by a short extensions, because the length of the payloads for those short extensions is known. This actually reduces the encoding length compared to the previous example. The following encoding is also valid:

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0xF1, 0x04, 0x45, 0x30, 0x65, 0x78, 0x03,
 0x00, 0x3C, 0x04, 0x45, 0x30, 0x65, 0x78, 0x32}
```

Figure 16: Encoding of indices 0...9 (24 bytes)

Here we have inserted a frame separator (ID=1, L=1, hex encoding 0x03) with a frame increment of zero (hex encoding 0x00). This is treated as padding and does not change which extensions get repeated.

A.11. Example 10

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0xF1, 0x04, 0x45, 0x30, 0x65, 0x78, 0x3C,
 0x05, 0x05, 0x45, 0x30, 0x65, 0x78, 0x32, 0x02,
 0x3F, 0x66}
```

Figure 17: Encoding of indices 0...10 (26 bytes)

Here we have added another short extension to the last frame. We cannot use L=0 on the final repeat indicator to skip a frame separator, because our repeated extensions included a long extension.

A.12. Example 11

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0xF1, 0x04, 0x45, 0x30, 0x65, 0x78, 0x3C,
 0x3F, 0x65, 0x04, 0x45, 0x30, 0x65, 0x78, 0x32,
 0x66}
```

Figure 18: Encoding of indices 0...11 (25 bytes)

However, if we add the short extension in both of the final two frames, we can use L=0 on the final repeat indicator, even though the subsequent short extensions have a payload byte. In this case, the final byte ("f", hex encoding 0x66) is the payload of that short extension in the final frame, and not part of the long extension, even though no length was encoded for the final long extension. This

is possible because we know in advance which short extensions are being repeated, so we know how many bytes to reserve for their payloads at the end. Again, we have added an extension and even another payload byte, but our encoding is shorter than the previous example.

A.13. Example 12

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0xF1, 0x04, 0x45, 0x30, 0x65, 0x78, 0x3C,
 0x3F, 0x65, 0x05, 0x05, 0x45, 0x30, 0x65, 0x78,
 0x32, 0x66, 0x02, 0xF0, 0x45, 0x30, 0x65, 0x78,
 0x61, 0x6D, 0x70, 0x6C, 0x65}
```

Figure 19: Encoding of indices 0...12 (37 bytes)

Now, we add another extension to the last frame. Even though the final repeated extension is a short extension, we cannot use L=0 on the repeat to skip a frame separator, because it also repeated a long extension. However, the following encoding is also valid:

```
{0x39, 0x61, 0x05, 0x62, 0x63, 0x3B, 0x64, 0x02,
 0x3A, 0xF1, 0x04, 0x45, 0x30, 0x65, 0x78, 0x3C,
 0x3F, 0x65, 0x05, 0x05, 0x45, 0x30, 0x65, 0x78,
 0x32, 0x66, 0x04, 0xF0, 0x45, 0x30, 0x65, 0x78,
 0x61, 0x6D, 0x70, 0x6C, 0x65}
```

Figure 20: Encoding of indices 0...12 (37 bytes)

Here, the final frame separator (ID=1, L=0, hex encoding 0x02) has been replaced by a repeat indicator (ID=2, L=0, hex encoding 0x04). Because there have been no new extensions to repeat since the previous repeat indicator, this merely increments the current frame index the same way that a frame separator would.

Authors' Addresses

Timothy B. Terriberry
Xiph.Org
United States of America
Email: tterribe@xiph.org

Jean-Marc Valin
Google
Canada
Email: jeanmarcv@google.com