Internet Engineering Task Force (IETF)
Request for Comments: 7541
Category: Standards Track
ISSN: 2070-1721

R. Peon
Google, Inc
H. Ruellan
Canon CRF
May 2015

HPACK: Header Compression for HTTP/2

Abstract

This specification defines HPACK, a compression format for efficiently representing HTTP header fields, to be used in HTTP/2.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741[1].

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7541[2].

Copyright Notice

Copyright (c) 2015 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 (http://trustee.ietf.org/license-info[3]) 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.

[1] https://www.rfc-editor.org/rfc/rfc5741.html#section-2
[2] http://www.rfc-editor.org/info/rfc7541
[3] http://trustee.ietf.org/license-info

Table of Contents

1 Introduction 4
1.1 Overview 4
1.2 Conventions 4
1.3 Terminology 4
2 Compression Process Overview 6
2.1 Header List Ordering 6
2.2 Encoding and Decoding Contexts 6
2.3 Indexing Tables 6
2.3.1 Static Table 6
2.3.2 Dynamic Table 6
2.3.3 Index Address Space 6
2.4 Header Field Representation 7
3 Header Block Decoding 8
3.1 Header Block Processing 8
3.2 Header Field Representation Processing 8
4 Dynamic Table Management 9
4.1 Calculating Table Size 9
4.2 Maximum Table Size 9
4.3 Entry Eviction When Dynamic Table Size Changes 9
4.4 Entry Eviction When Adding New Entries 9
5 Primitive Type Representations 10
5.1 Integer Representation 10
5.2 String Literal Representation 11
6 Binary Format 12
6.1 Indexed Header Field Representation 12
6.2 Literal Header Field Representation 12
6.2.1 Literal Header Field with Incremental Indexing 12
6.2.2 Literal Header Field without Indexing 13
6.2.3 Literal Header Field Never Indexed 14
6.3 Dynamic Table Size Update 14
7 Security Considerations 16
7.1 Probing Dynamic Table State 16
7.1.1 Applicability to HPACK and HTTP 16
7.1.2 Mitigation 16
7.1.3 Never-Indexed Literals 17
7.2 Static Huffman Encoding 17
7.3 Memory Consumption 18

7.4 Implementation Limits....18

8 References....19

8.1 Normative References....19

8.2 Informative References....19

Appendix A Static Table Definition....20

Appendix B Huffman Code....22

Appendix C Examples....24

C.1 Integer Representation Examples....24
C.1.1 Example 1: Encoding 10 Using a 5-Bit Prefix....24
C.1.2 Example 2: Encoding 1337 Using a 5-Bit Prefix....24
C.1.3 Example 3: Encoding 42 Starting at an Octet Boundary....24

C.2 Header Field Representation Examples....24
C.2.1 Literal Header Field with Indexing....25
C.2.2 Literal Header Field without Indexing....25
C.2.3 Literal Header Field Never Indexed....26
C.2.4 Indexed Header Field....26

C.3 Request Examples without Huffman Coding....27
C.3.1 First Request....27
C.3.2 Second Request....28
C.3.3 Third Request....29

C.4 Request Examples with Huffman Coding....30
C.4.1 First Request....30
C.4.2 Second Request....31
C.4.3 Third Request....32

C.5 Response Examples without Huffman Coding....33
C.5.1 First Response....33
C.5.2 Second Response....34
C.5.3 Third Response....35

C.6 Response Examples with Huffman Coding....38
C.6.1 First Response....38
C.6.2 Second Response....40
C.6.3 Third Response....41

Authors' Addresses....45

1. Introduction

In HTTP/1.1 (see [RFC7230]), header fields are not compressed. As web pages have grown to require dozens to hundreds of requests, the redundant header fields in these requests unnecessarily consume bandwidth, measurably increasing latency.

SPDY [SPDY] initially addressed this redundancy by compressing header fields using the DEFLATE [DEFLATE] format, which proved very effective at efficiently representing the redundant header fields. However, that approach exposed a security risk as demonstrated by the CRIME (Compression Ratio Info-leak Made Easy) attack (see [CRIME]).

This specification defines HPACK, a new compressor that eliminates redundant header fields, limits vulnerability to known security attacks, and has a bounded memory requirement for use in constrained environments. Potential security concerns for HPACK are described in Section 7.

The HPACK format is intentionally simple and inflexible. Both characteristics reduce the risk of interoperability or security issues due to implementation error. No extensibility mechanisms are defined; changes to the format are only possible by defining a complete replacement.

1.1. Overview

The format defined in this specification treats a list of header fields as an ordered collection of name-value pairs that can include duplicate pairs. Names and values are considered to be opaque sequences of octets, and the order of header fields is preserved after being compressed and decompressed.

Encoding is informed by header field tables that map header fields to indexed values. These header field tables can be incrementally updated as new header fields are encoded or decoded.

In the encoded form, a header field is represented either literally or as a reference to a header field in one of the header field tables. Therefore, a list of header fields can be encoded using a mixture of references and literal values.

Literal values are either encoded directly or use a static Huffman code.

The encoder is responsible for deciding which header fields to insert as new entries in the header field tables. The decoder executes the modifications to the header field tables prescribed by the encoder, reconstructing the list of header fields in the process. This enables decoders to remain simple and interoperate with a wide variety of encoders.

Examples illustrating the use of these different mechanisms to represent header fields are available in Appendix C.

1.2. Conventions

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 RFC 2119 [RFC2119].

All numeric values are in network byte order. Values are unsigned unless otherwise indicated. Literal values are provided in decimal or hexadecimal as appropriate.

1.3. Terminology

This specification uses the following terms:

Header Field:
: A name-value pair. Both the name and value are treated as opaque sequences of octets.

Dynamic Table:
: The dynamic table (see Section 2.3.2) is a table that associates stored header fields with index values.

This table is dynamic and specific to an encoding or decoding context.

Static Table:
: The static table (see Section 2.3.1) is a table that statically associates header fields that occur frequently with index values. This table is ordered, read-only, always accessible, and it may be shared amongst all encoding or decoding contexts.

Header List:
: A header list is an ordered collection of header fields that are encoded jointly and can contain duplicate header fields. A complete list of header fields contained in an HTTP/2 header block is a header list.

Header Field Representation:
: A header field can be represented in encoded form either as a literal or as an index (see Section 2.4).

Header Block:
: An ordered list of header field representations, which, when decoded, yields a complete header list.

2. Compression Process Overview

This specification does not describe a specific algorithm for an encoder. Instead, it defines precisely how a decoder is expected to operate, allowing encoders to produce any encoding that this definition permits.

2.1. Header List Ordering

HPACK preserves the ordering of header fields inside the header list. An encoder MUST order header field representations in the header block according to their ordering in the original header list. A decoder MUST order header fields in the decoded header list according to their ordering in the header block.

2.2. Encoding and Decoding Contexts

To decompress header blocks, a decoder only needs to maintain a dynamic table (see Section 2.3.2) as a decoding context. No other dynamic state is needed.

When used for bidirectional communication, such as in HTTP, the encoding and decoding dynamic tables maintained by an endpoint are completely independent, i.e., the request and response dynamic tables are separate.

2.3. Indexing Tables

HPACK uses two tables for associating header fields to indexes. The static table (see Section 2.3.1) is predefined and contains common header fields (most of them with an empty value). The dynamic table (see Section 2.3.2) is dynamic and can be used by the encoder to index header fields repeated in the encoded header lists.

These two tables are combined into a single address space for defining index values (see Section 2.3.3).

2.3.1. Static Table

The static table consists of a predefined static list of header fields. Its entries are defined in Appendix A.

2.3.2. Dynamic Table

The dynamic table consists of a list of header fields maintained in first-in, first-out order. The first and newest entry in a dynamic table is at the lowest index, and the oldest entry of a dynamic table is at the highest index.

The dynamic table is initially empty. Entries are added as each header block is decompressed.

The dynamic table can contain duplicate entries (i.e., entries with the same name and same value). Therefore, duplicate entries MUST NOT be treated as an error by a decoder.

The encoder decides how to update the dynamic table and as such can control how much memory is used by the dynamic table. To limit the memory requirements of the decoder, the dynamic table size is strictly bounded (see Section 4.2).

The decoder updates the dynamic table during the processing of a list of header field representations (see Section 3.2).

2.3.3. Index Address Space

The static table and the dynamic table are combined into a single index address space.

Indices between 1 and the length of the static table (inclusive) refer to elements in the static table (see Section 2.3.1).

Indices strictly greater than the length of the static table refer to elements in the dynamic table (see Section 2.3.2). The length of the static table is subtracted to find the index into the dynamic table.

Indices strictly greater than the sum of the lengths of both tables MUST be treated as a decoding error.

For a static table size of s and a dynamic table size of k, the following diagram shows the entire valid index address space.

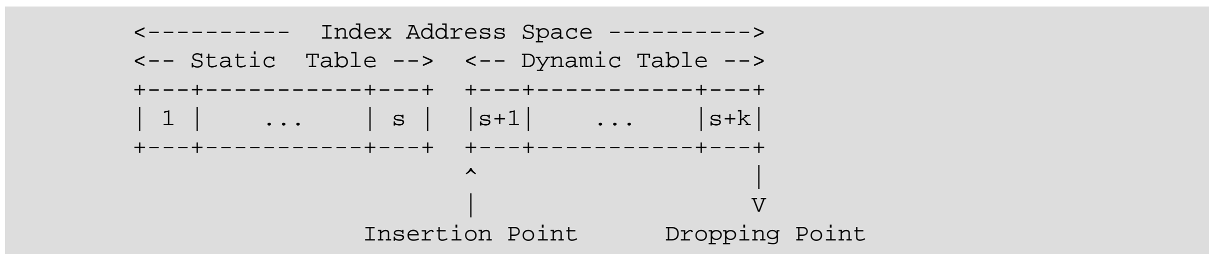


Figure 1: Index Address Space

2.4. Header Field Representation

An encoded header field can be represented either as an index or as a literal.

An indexed representation defines a header field as a reference to an entry in either the static table or the dynamic table (see Section 6.1).

A literal representation defines a header field by specifying its name and value. The header field name can be represented literally or as a reference to an entry in either the static table or the dynamic table. The header field value is represented literally.

Three different literal representations are defined:

- A literal representation that adds the header field as a new entry at the beginning of the dynamic table (see Section 6.2.1).
- A literal representation that does not add the header field to the dynamic table (see Section 6.2.2).
- A literal representation that does not add the header field to the dynamic table, with the additional stipulation that this header field always use a literal representation, in particular when re-encoded by an intermediary (see Section 6.2.3). This representation is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1.3 for more details).

The selection of one of these literal representations can be guided by security considerations, in order to protect sensitive header field values (see Section 7.1).

The literal representation of a header field name or of a header field value can encode the sequence of octets either directly or using a static Huffman code (see Section 5.2).

3. Header Block Decoding

3.1. Header Block Processing

A decoder processes a header block sequentially to reconstruct the original header list.

A header block is the concatenation of header field representations. The different possible header field representations are described in Section 6.

Once a header field is decoded and added to the reconstructed header list, the header field cannot be removed. A header field added to the header list can be safely passed to the application.

By passing the resulting header fields to the application, a decoder can be implemented with minimal transitory memory commitment in addition to the memory required for the dynamic table.

3.2. Header Field Representation Processing

The processing of a header block to obtain a header list is defined in this section. To ensure that the decoding will successfully produce a header list, a decoder MUST obey the following rules.

All the header field representations contained in a header block are processed in the order in which they appear, as specified below. Details on the formatting of the various header field representations and some additional processing instructions are found in Section 6.

An *indexed representation* entails the following actions:

- The header field corresponding to the referenced entry in either the static table or dynamic table is appended to the decoded header list.

A *literal representation* that is *not added* to the dynamic table entails the following action:

- The header field is appended to the decoded header list.

A *literal representation* that is *added* to the dynamic table entails the following actions:

- The header field is appended to the decoded header list.
- The header field is inserted at the beginning of the dynamic table. This insertion could result in the eviction of previous entries in the dynamic table (see Section 4.4).

4. Dynamic Table Management

To limit the memory requirements on the decoder side, the dynamic table is constrained in size.

4.1. Calculating Table Size

The size of the dynamic table is the sum of the size of its entries.

The size of an entry is the sum of its name's length in octets (as defined in Section 5.2), its value's length in octets, and 32.

The size of an entry is calculated using the length of its name and value without any Huffman encoding applied.

> **Note:** The additional 32 octets account for an estimated overhead associated with an entry. For example, an entry structure using two 64-bit pointers to reference the name and the value of the entry and two 64-bit integers for counting the number of references to the name and value would have 32 octets of overhead.

4.2. Maximum Table Size

Protocols that use HPACK determine the maximum size that the encoder is permitted to use for the dynamic table. In HTTP/2, this value is determined by the SETTINGS_HEADER_TABLE_SIZE setting (see Section 6.5.2 of [HTTP2]).

An encoder can choose to use less capacity than this maximum size (see Section 6.3), but the chosen size MUST stay lower than or equal to the maximum set by the protocol.

A change in the maximum size of the dynamic table is signaled via a dynamic table size update (see Section 6.3). This dynamic table size update MUST occur at the beginning of the first header block following the change to the dynamic table size. In HTTP/2, this follows a settings acknowledgment (see Section 6.5.3 of [HTTP2]).

Multiple updates to the maximum table size can occur between the transmission of two header blocks. In the case that this size is changed more than once in this interval, the smallest maximum table size that occurs in that interval MUST be signaled in a dynamic table size update. The final maximum size is always signaled, resulting in at most two dynamic table size updates. This ensures that the decoder is able to perform eviction based on reductions in dynamic table size (see Section 4.3).

This mechanism can be used to completely clear entries from the dynamic table by setting a maximum size of 0, which can subsequently be restored.

4.3. Entry Eviction When Dynamic Table Size Changes

Whenever the maximum size for the dynamic table is reduced, entries are evicted from the end of the dynamic table until the size of the dynamic table is less than or equal to the maximum size.

4.4. Entry Eviction When Adding New Entries

Before a new entry is added to the dynamic table, entries are evicted from the end of the dynamic table until the size of the dynamic table is less than or equal to (maximum size - new entry size) or until the table is empty.

If the size of the new entry is less than or equal to the maximum size, that entry is added to the table. It is not an error to attempt to add an entry that is larger than the maximum size; an attempt to add an entry larger than the maximum size causes the table to be emptied of all existing entries and results in an empty table.

A new entry can reference the name of an entry in the dynamic table that will be evicted when adding this new entry into the dynamic table. Implementations are cautioned to avoid deleting the referenced name if the referenced entry is evicted from the dynamic table prior to inserting the new entry.

5. Primitive Type Representations

HPACK encoding uses two primitive types: unsigned variable-length integers and strings of octets.

5.1. Integer Representation

Integers are used to represent name indexes, header field indexes, or string lengths. An integer representation can start anywhere within an octet. To allow for optimized processing, an integer representation always finishes at the end of an octet.

An integer is represented in two parts: a prefix that fills the current octet and an optional list of octets that are used if the integer value does not fit within the prefix. The number of bits of the prefix (called N) is a parameter of the integer representation.

If the integer value is small enough, i.e., strictly less than 2^N-1, it is encoded within the N-bit prefix.

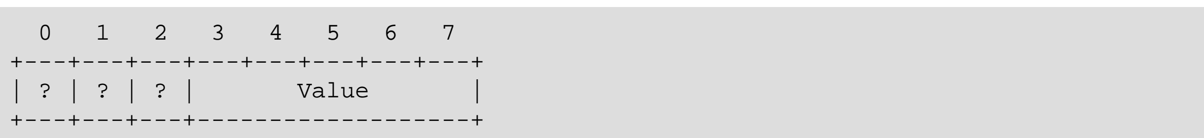


Figure 2: Integer Value Encoded within the Prefix (Shown for N = 5)

Otherwise, all the bits of the prefix are set to 1, and the value, decreased by 2^N-1, is encoded using a list of one or more octets. The most significant bit of each octet is used as a continuation flag: its value is set to 1 except for the last octet in the list. The remaining bits of the octets are used to encode the decreased value.

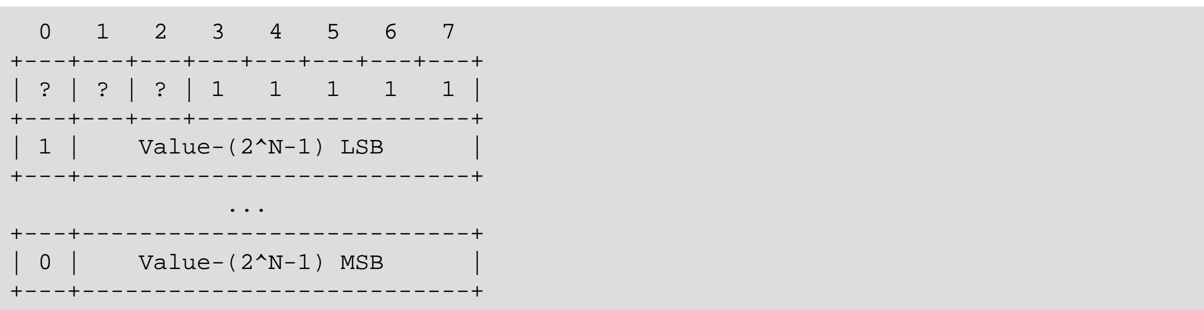


Figure 3: Integer Value Encoded after the Prefix (Shown for N = 5)

Decoding the integer value from the list of octets starts by reversing the order of the octets in the list. Then, for each octet, its most significant bit is removed. The remaining bits of the octets are concatenated, and the resulting value is increased by 2^N-1 to obtain the integer value.

The prefix size, N, is always between 1 and 8 bits. An integer starting at an octet boundary will have an 8-bit prefix.

Pseudocode to represent an integer I is as follows:

```
if I < 2^N - 1, encode I on N bits
else
    encode (2^N - 1) on N bits
    I = I - (2^N - 1)
    while I >= 128
         encode (I % 128 + 128) on 8 bits
         I = I / 128
    encode I on 8 bits
```

Pseudocode to decode an integer I is as follows:

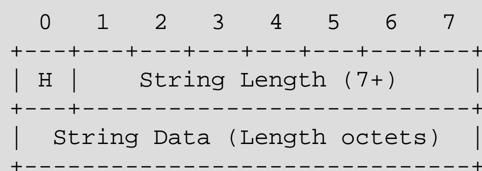
```
decode I from the next N bits
if I < 2^N - 1, return I
else
    M = 0
    repeat
        B = next octet
        I = I + (B & 127) * 2^M
        M = M + 7
    while B & 128 == 128
    return I
```

Examples illustrating the encoding of integers are available in Appendix C.1.

This integer representation allows for values of indefinite size. It is also possible for an encoder to send a large number of zero values, which can waste octets and could be used to overflow integer values. Integer encodings that exceed implementation limits — in value or octet length — MUST be treated as decoding errors. Different limits can be set for each of the different uses of integers, based on implementation constraints.

5.2. String Literal Representation

Header field names and header field values can be represented as string literals. A string literal is encoded as a sequence of octets, either by directly encoding the string literal's octets or by using a Huffman code (see [HUFFMAN]).



```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| H |    String Length (7+)     |
+---+---------------------------+
|  String Data (Length octets)  |
+-------------------------------+
```

Figure 4: String Literal Representation

A string literal representation contains the following fields:

H: A one-bit flag, H, indicating whether or not the octets of the string are Huffman encoded.

String Length: The number of octets used to encode the string literal, encoded as an integer with a 7-bit prefix (see Section 5.1).

String Data: The encoded data of the string literal. If H is '0', then the encoded data is the raw octets of the string literal. If H is '1', then the encoded data is the Huffman encoding of the string literal.

String literals that use Huffman encoding are encoded with the Huffman code defined in Appendix B (see examples for requests in Appendix C.4 and for responses in Appendix C.6). The encoded data is the bitwise concatenation of the codes corresponding to each octet of the string literal.

As the Huffman-encoded data doesn't always end at an octet boundary, some padding is inserted after it, up to the next octet boundary. To prevent this padding from being misinterpreted as part of the string literal, the most significant bits of the code corresponding to the EOS (end-of-string) symbol are used.

Upon decoding, an incomplete code at the end of the encoded data is to be considered as padding and discarded. A padding strictly longer than 7 bits MUST be treated as a decoding error. A padding not corresponding to the most significant bits of the code for the EOS symbol MUST be treated as a decoding error. A Huffman-encoded string literal containing the EOS symbol MUST be treated as a decoding error.

6. Binary Format

This section describes the detailed format of each of the different header field representations and the dynamic table size update instruction.

6.1. Indexed Header Field Representation

An indexed header field representation identifies an entry in either the static table or the dynamic table (see Section 2.3).

An indexed header field representation causes a header field to be added to the decoded header list, as described in Section 3.2.

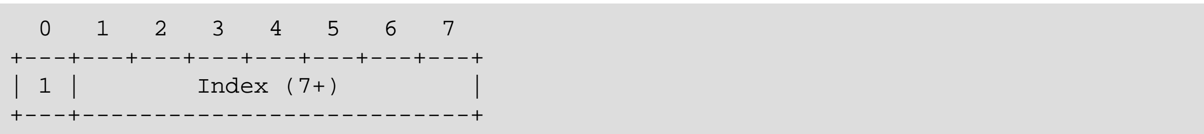


Figure 5: Indexed Header Field

An indexed header field starts with the '1' 1-bit pattern, followed by the index of the matching header field, represented as an integer with a 7-bit prefix (see Section 5.1).

The index value of 0 is not used. It MUST be treated as a decoding error if found in an indexed header field representation.

6.2. Literal Header Field Representation

A literal header field representation contains a literal header field value. Header field names are provided either as a literal or by reference to an existing table entry, either from the static table or the dynamic table (see Section 2.3).

This specification defines three forms of literal header field representations: with indexing, without indexing, and never indexed.

6.2.1. Literal Header Field with Incremental Indexing

A literal header field with incremental indexing representation results in appending a header field to the decoded header list and inserting it as a new entry into the dynamic table.

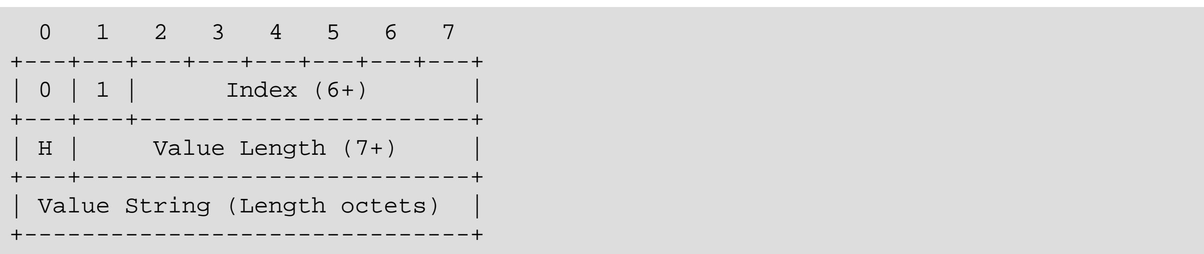
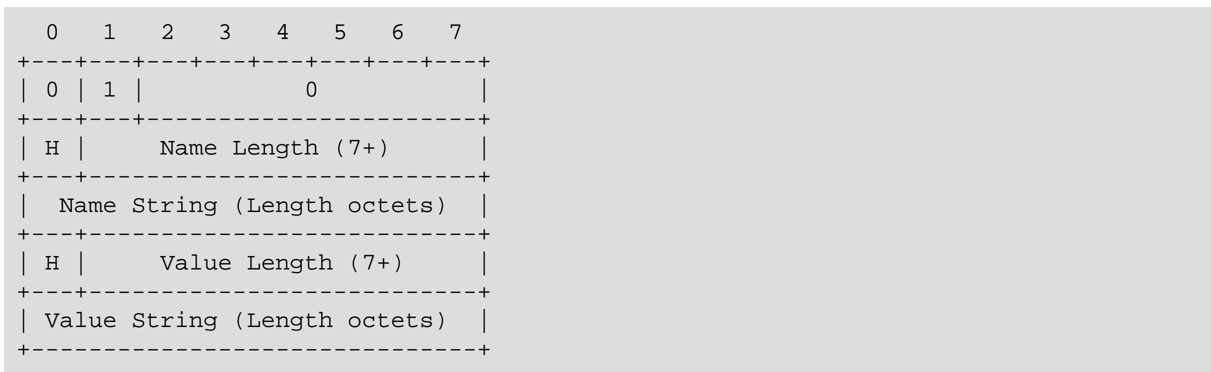


Figure 6: Literal Header Field with Incremental Indexing — Indexed Name



```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 1 |           0           |
+---+---+-----------------------+
| H |     Name Length (7+)      |
+---+---------------------------+
|  Name String (Length octets)  |
+---+---------------------------+
| H |     Value Length (7+)     |
+---+---------------------------+
| Value String (Length octets)  |
+-------------------------------+
```

Figure 7: Literal Header Field with Incremental Indexing — New Name

A literal header field with incremental indexing representation starts with the '01' 2-bit pattern.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with a 6-bit prefix (see Section 5.1). This value is always non-zero.

Otherwise, the header field name is represented as a string literal (see Section 5.2). A value 0 is used in place of the 6-bit index, followed by the header field name.

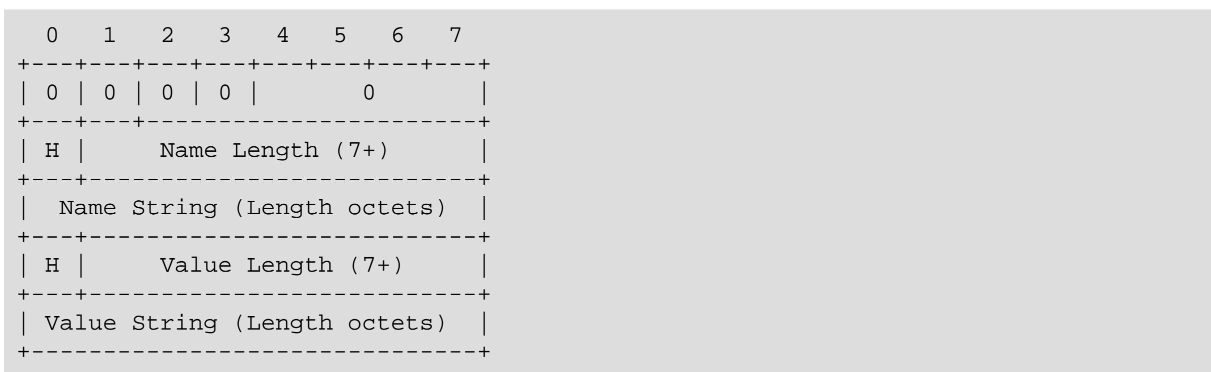
Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2).

6.2.2. Literal Header Field without Indexing

A literal header field without indexing representation results in appending a header field to the decoded header list without altering the dynamic table.

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 |  Index (4+)   |
+---+---+-----------------------+
| H |     Value Length (7+)     |
+---+---------------------------+
| Value String (Length octets)  |
+-------------------------------+
```

Figure 8: Literal Header Field without Indexing — Indexed Name



```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 |       0       |
+---+---+-----------------------+
| H |     Name Length (7+)      |
+---+---------------------------+
|  Name String (Length octets)  |
+---+---------------------------+
| H |     Value Length (7+)     |
+---+---------------------------+
| Value String (Length octets)  |
+-------------------------------+
```

Figure 9: Literal Header Field without Indexing — New Name

A literal header field without indexing representation starts with the '0000' 4-bit pattern.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with a 4-bit prefix (see Section 5.1). This value is always non-zero.

Otherwise, the header field name is represented as a string literal (see Section 5.2). A value 0 is used in place of the 4-bit index, followed by the header field name.

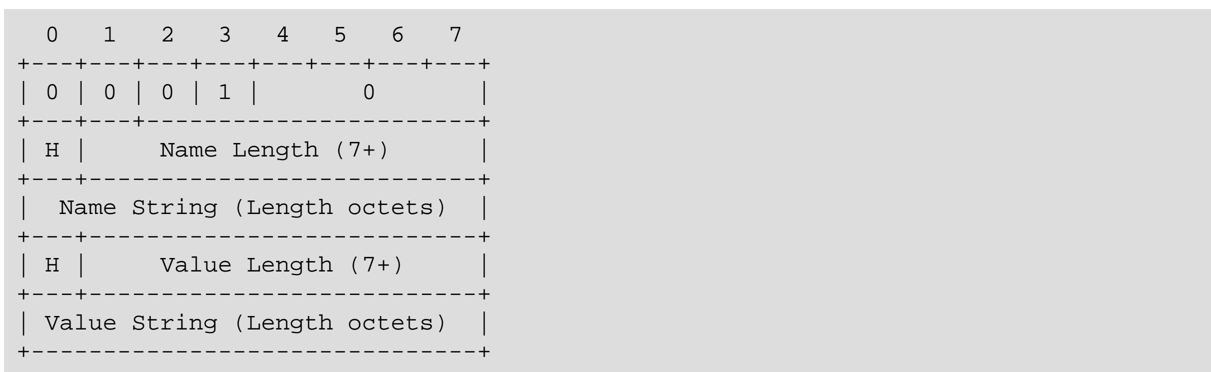
Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2).

6.2.3. Literal Header Field Never Indexed

A literal header field never-indexed representation results in appending a header field to the decoded header list without altering the dynamic table. Intermediaries MUST use the same representation for encoding this header field.

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 1 |  Index (4+)   |
+---+---+-----------------------+
| H |     Value Length (7+)     |
+---+---------------------------+
| Value String (Length octets)  |
+-------------------------------+
```

Figure 10: Literal Header Field Never Indexed — Indexed Name



```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 1 |       0       |
+---+---+-----------------------+
| H |     Name Length (7+)      |
+---+---------------------------+
|  Name String (Length octets)  |
+---+---------------------------+
| H |     Value Length (7+)     |
+---+---------------------------+
| Value String (Length octets)  |
+-------------------------------+
```

Figure 11: Literal Header Field Never Indexed — New Name

A literal header field never-indexed representation starts with the '0001' 4-bit pattern.

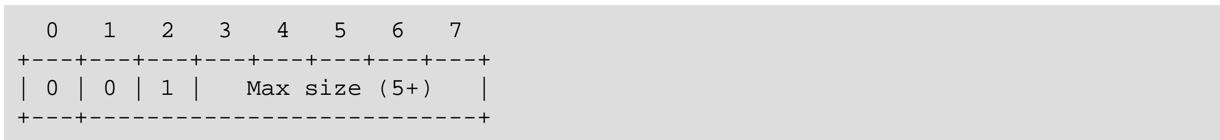
When a header field is represented as a literal header field never indexed, it MUST always be encoded with this specific literal representation. In particular, when a peer sends a header field that it received represented as a literal header field never indexed, it MUST use the same representation to forward this header field.

This representation is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1 for more details).

The encoding of the representation is identical to the literal header field without indexing (see Section 6.2.2).

6.3. Dynamic Table Size Update

A dynamic table size update signals a change to the size of the dynamic table.



```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 1 |   Max size (5+)   |
+---+---------------------------+
```

Figure 12: Maximum Dynamic Table Size Change

A dynamic table size update starts with the '001' 3-bit pattern, followed by the new maximum size, represented as an integer with a 5-bit prefix (see Section 5.1).

The new maximum size MUST be lower than or equal to the limit determined by the protocol using HPACK. A value that exceeds this limit MUST be treated as a decoding error. In HTTP/2, this limit is the last value of the SETTINGS_HEADER_TABLE_SIZE parameter (see Section 6.5.2 of [HTTP2]) received from the decoder and acknowledged by the encoder (see Section 6.5.3 of [HTTP2]).

Reducing the maximum size of the dynamic table can cause entries to be evicted (see Section 4.3).

7. Security Considerations

This section describes potential areas of security concern with HPACK:

- Use of compression as a length-based oracle for verifying guesses about secrets that are compressed into a shared compression context.
- Denial of service resulting from exhausting processing or memory capacity at a decoder.

7.1. Probing Dynamic Table State

HPACK reduces the length of header field encodings by exploiting the redundancy inherent in protocols like HTTP. The ultimate goal of this is to reduce the amount of data that is required to send HTTP requests or responses.

The compression context used to encode header fields can be probed by an attacker who can both define header fields to be encoded and transmitted and observe the length of those fields once they are encoded. When an attacker can do both, they can adaptively modify requests in order to confirm guesses about the dynamic table state. If a guess is compressed into a shorter length, the attacker can observe the encoded length and infer that the guess was correct.

This is possible even over the Transport Layer Security (TLS) protocol (see [TLS12]), because while TLS provides confidentiality protection for content, it only provides a limited amount of protection for the length of that content.

> **Note:** Padding schemes only provide limited protection against an attacker with these capabilities, potentially only forcing an increased number of guesses to learn the length associated with a given guess. Padding schemes also work directly against compression by increasing the number of bits that are transmitted.

Attacks like CRIME [CRIME] demonstrated the existence of these general attacker capabilities. The specific attack exploited the fact that DEFLATE [DEFLATE] removes redundancy based on prefix matching. This permitted the attacker to confirm guesses a character at a time, reducing an exponential-time attack into a linear-time attack.

7.1.1. Applicability to HPACK and HTTP

HPACK mitigates but does not completely prevent attacks modeled on CRIME [CRIME] by forcing a guess to match an entire header field value rather than individual characters. Attackers can only learn whether a guess is correct or not, so they are reduced to brute-force guesses for the header field values.

The viability of recovering specific header field values therefore depends on the entropy of values. As a result, values with high entropy are unlikely to be recovered successfully. However, values with low entropy remain vulnerable.

Attacks of this nature are possible any time that two mutually distrustful entities control requests or responses that are placed onto a single HTTP/2 connection. If the shared HPACK compressor permits one entity to add entries to the dynamic table and the other to access those entries, then the state of the table can be learned.

Having requests or responses from mutually distrustful entities occurs when an intermediary either:

- sends requests from multiple clients on a single connection toward an origin server, or
- takes responses from multiple origin servers and places them on a shared connection toward a client.

Web browsers also need to assume that requests made on the same connection by different web origins [ORIGIN] are made by mutually distrustful entities.

7.1.2. Mitigation

Users of HTTP that require confidentiality for header fields can use values with entropy sufficient to make guessing infeasible. However, this is impractical as a general solution because it forces all users of HTTP to take steps to mitigate attacks. It would impose new constraints on how HTTP is used.

Rather than impose constraints on users of HTTP, an implementation of HPACK can instead constrain how compression is applied in order to limit the potential for dynamic table probing.

An ideal solution segregates access to the dynamic table based on the entity that is constructing header fields. Header field values that are added to the table are attributed to an entity, and only the entity that created a particular value can extract that value.

To improve compression performance of this option, certain entries might be tagged as being public. For example, a web browser might make the values of the Accept-Encoding header field available in all requests.

An encoder without good knowledge of the provenance of header fields might instead introduce a penalty for a header field with many different values, such that a large number of attempts to guess a header field value results in the header field no longer being compared to the dynamic table entries in future messages, effectively preventing further guesses.

> **Note:** Simply removing entries corresponding to the header field from the dynamic table can be ineffectual if the attacker has a reliable way of causing values to be reinstalled. For example, a request to load an image in a web browser typically includes the Cookie header field (a potentially highly valued target for this sort of attack), and web sites can easily force an image to be loaded, thereby refreshing the entry in the dynamic table.

This response might be made inversely proportional to the length of the header field value. Marking a header field as not using the dynamic table anymore might occur for shorter values more quickly or with higher probability than for longer values.

7.1.3. Never-Indexed Literals

Implementations can also choose to protect sensitive header fields by not compressing them and instead encoding their value as literals.

Refusing to generate an indexed representation for a header field is only effective if compression is avoided on all hops. The never-indexed literal (see Section 6.2.3) can be used to signal to intermediaries that a particular value was intentionally sent as a literal.

An intermediary MUST NOT re-encode a value that uses the never-indexed literal representation with another representation that would index it. If HPACK is used for re-encoding, the never-indexed literal representation MUST be used.

The choice to use a never-indexed literal representation for a header field depends on several factors. Since HPACK doesn't protect against guessing an entire header field value, short or low-entropy values are more readily recovered by an adversary. Therefore, an encoder might choose not to index values with low entropy.

An encoder might also choose not to index values for header fields that are considered to be highly valuable or sensitive to recovery, such as the Cookie or Authorization header fields.

On the contrary, an encoder might prefer indexing values for header fields that have little or no value if they were exposed. For instance, a User-Agent header field does not commonly vary between requests and is sent to any server. In that case, confirmation that a particular User-Agent value has been used provides little value.

Note that these criteria for deciding to use a never-indexed literal representation will evolve over time as new attacks are discovered.

7.2. Static Huffman Encoding

There is no currently known attack against a static Huffman encoding. A study has shown that using a static Huffman encoding table created an information leakage; however, this same study concluded that an attacker could not take advantage of this information leakage to recover any meaningful amount of information (see [PETAL]).

7.3. Memory Consumption

An attacker can try to cause an endpoint to exhaust its memory. HPACK is designed to limit both the peak and state amounts of memory allocated by an endpoint.

The amount of memory used by the compressor is limited by the protocol using HPACK through the definition of the maximum size of the dynamic table. In HTTP/2, this value is controlled by the decoder through the setting parameter SETTINGS_HEADER_TABLE_SIZE (see Section 6.5.2 of [HTTP2]). This limit takes into account both the size of the data stored in the dynamic table, plus a small allowance for overhead.

A decoder can limit the amount of state memory used by setting an appropriate value for the maximum size of the dynamic table. In HTTP/2, this is realized by setting an appropriate value for the SETTINGS_HEADER_TABLE_SIZE parameter. An encoder can limit the amount of state memory it uses by signaling a lower dynamic table size than the decoder allows (see Section 6.3).

The amount of temporary memory consumed by an encoder or decoder can be limited by processing header fields sequentially. An implementation does not need to retain a complete list of header fields. Note, however, that it might be necessary for an application to retain a complete header list for other reasons; even though HPACK does not force this to occur, application constraints might make this necessary.

7.4. Implementation Limits

An implementation of HPACK needs to ensure that large values for integers, long encoding for integers, or long string literals do not create security weaknesses.

An implementation has to set a limit for the values it accepts for integers, as well as for the encoded length (see Section 5.1). In the same way, it has to set a limit to the length it accepts for string literals (see Section 5.2).

8. References

8.1. Normative References

[HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc>.

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

[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, <http://www.rfc-editor.org/info/rfc>.

8.2. Informative References

[CANONICAL] Schwartz, E. and B. Kallick, "Generating a canonical prefix encoding", Communications of the ACM, Volume 7 Issue 3, pp. 166-169, March 1964, <https://dl.acm.org/citation.cfm?id=363991>.

[CRIME] Wikipedia, "CRIME", May 2015, <http://en.wikipedia.org/w/index.php?title=CRIME&oldid=660948120>.

[DEFLATE] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996, <http://www.rfc-editor.org/info/rfc>.

[HUFFMAN] Huffman, D., "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the Institute of Radio Engineers, Volume 40, Number 9, pp. 1098-1101, September 1952, <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4051119>.

[ORIGIN] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <http://www.rfc-editor.org/info/rfc>.

[PETAL] Tan, J. and J. Nahata, "PETAL: Preset Encoding Table Information Leakage", April 2013, <http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-13-106.pdf>.

[SPDY] Belshe, M. and R. Peon, "SPDY Protocol", Work in Progress, draft-mbelshe-httpbis-spdy-00, February 2012.

[TLS12] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc>.

Appendix A. Static Table Definition

The static table (see Section 2.3.1) consists in a predefined and unchangeable list of header fields.

The static table was created from the most frequent header fields used by popular web sites, with the addition of HTTP/2-specific pseudo-header fields (see Section 8.1.2.1 of [HTTP2]). For header fields with a few frequent values, an entry was added for each of these frequent values. For other header fields, an entry was added with an empty value.

Table 1 lists the predefined header fields that make up the static table and gives the index of each entry.

Index	Header Name	Header Value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
9	:status	204
10	:status	206
11	:status	304
12	:status	400
13	:status	404
14	:status	500
15	accept-charset	
16	accept-encoding	gzip, deflate
17	accept-language	
18	accept-ranges	
19	accept	
20	access-control-allow-origin	
21	age	
22	allow	
23	authorization	
24	cache-control	
25	content-disposition	
26	content-encoding	
27	content-language	
28	content-length	
29	content-location	
30	content-range	
31	content-type	
32	cookie	
33	date	
34	etag	
35	expect	
36	expires	
37	from	
38	host	
39	if-match	
40	if-modified-since	
41	if-none-match	
42	if-range	
43	if-unmodified-since	

Index	Header Name	Header Value
44	last-modified	
45	link	
46	location	
47	max-forwards	
48	proxy-authenticate	
49	proxy-authorization	
50	range	
51	referer	
52	refresh	
53	retry-after	
54	server	
55	set-cookie	
56	strict-transport-security	
57	transfer-encoding	
58	user-agent	
59	vary	
60	via	
61	www-authenticate	

Table 1: Static Table Entries

Appendix B. Huffman Code

The following Huffman code is used when encoding string literals with a Huffman coding (see Section 5.2).

This Huffman code was generated from statistics obtained on a large sample of HTTP headers. It is a canonical Huffman code (see [CANONICAL]) with some tweaking to ensure that no symbol has a unique code length.

Each row in the table defines the code used to represent a symbol:

sym: The symbol to be represented. It is the decimal value of an octet, possibly prepended with its ASCII representation. A specific symbol, "EOS", is used to indicate the end of a string literal.

code as bits: The Huffman code for the symbol represented as a base-2 integer, aligned on the most significant bit (MSB).

code as hex: The Huffman code for the symbol, represented as a hexadecimal integer, aligned on the least significant bit (LSB).

len: The number of bits for the code representing the symbol.

As an example, the code for the symbol 47 (corresponding to the ASCII character "/") consists in the 6 bits "0", "1", "1", "0", "0", "0". This corresponds to the value 0x18 (in hexadecimal) encoded in 6 bits.

```
                                                        code
                          code as bits                 as hex   len
        sym              aligned to MSB                aligned   in
                                                       to LSB   bits
       (  0)  |11111111|11000                             1ff8  [13]
       (  1)  |11111111|11111111|1011000                7fffd8  [23]
       (  2)  |11111111|11111111|11111110|0010         fffffe2  [28]
       (  3)  |11111111|11111111|11111110|0011         fffffe3  [28]
       (  4)  |11111111|11111111|11111110|0100         fffffe4  [28]
       (  5)  |11111111|11111111|11111110|0101         fffffe5  [28]
       (  6)  |11111111|11111111|11111110|0110         fffffe6  [28]
       (  7)  |11111111|11111111|11111110|0111         fffffe7  [28]
       (  8)  |11111111|11111111|11111110|1000         fffffe8  [28]
       (  9)  |11111111|11111111|11101010               ffffea  [24]
       ( 10)  |11111111|11111111|11111111|111100      3ffffffc  [30]
       ( 11)  |11111111|11111111|11111110|1001         fffffe9  [28]
       ( 12)  |11111111|11111111|11111110|1010         fffffea  [28]
       ( 13)  |11111111|11111111|11111111|111101      3ffffffd  [30]
       ( 14)  |11111111|11111111|11111110|1011         fffffeb  [28]
       ( 15)  |11111111|11111111|11111110|1100         fffffec  [28]
       ( 16)  |11111111|11111111|11111110|1101         fffffed  [28]
       ( 17)  |11111111|11111111|11111110|1110         fffffee  [28]
       ( 18)  |11111111|11111111|11111110|1111         fffffef  [28]
       ( 19)  |11111111|11111111|11111111|0000         ffffff0  [28]
       ( 20)  |11111111|11111111|11111111|0001         ffffff1  [28]
       ( 21)  |11111111|11111111|11111111|0010         ffffff2  [28]
       ( 22)  |11111111|11111111|11111111|111110      3ffffffe  [30]
       ( 23)  |11111111|11111111|11111111|0011         ffffff3  [28]
       ( 24)  |11111111|11111111|11111111|0100         ffffff4  [28]
       ( 25)  |11111111|11111111|11111111|0101         ffffff5  [28]
       ( 26)  |11111111|11111111|11111111|0110         ffffff6  [28]
       ( 27)  |11111111|11111111|11111111|0111         ffffff7  [28]
       ( 28)  |11111111|11111111|11111111|1000         ffffff8  [28]
       ( 29)  |11111111|11111111|11111111|1001         ffffff9  [28]
       ( 30)  |11111111|11111111|11111111|1010         ffffffa  [28]
       ( 31)  |11111111|11111111|11111111|1011         ffffffb  [28]
   ' ' ( 32)  |010100                                       14  [ 6]
   '!' ( 33)  |11111110|00                                 3f8  [10]
   '"' ( 34)  |11111110|01                                 3f9  [10]
   '#' ( 35)  |11111111|1010                               ffa  [12]
   '$' ( 36)  |11111111|11001                             1ff9  [13]
   '%' ( 37)  |010101                                       15  [ 6]
   '&' ( 38)  |11111000                                     f8  [ 8]
   ''' ( 39)  |11111111|010                                7fa  [11]
   '(' ( 40)  |11111110|10                                 3fa  [10]
   ')' ( 41)  |11111110|11                                 3fb  [10]
   '*' ( 42)  |11111001                                     f9  [ 8]
   '+' ( 43)  |11111111|011                                7fb  [11]
   ',' ( 44)  |11111010                                     fa  [ 8]
   '-' ( 45)  |010110                                       16  [ 6]
   '.' ( 46)  |010111                                       17  [ 6]
   '/' ( 47)  |011000                                       18  [ 6]
   '0' ( 48)  |00000                                         0  [ 5]
   '1' ( 49)  |00001                                         1  [ 5]
   '2' ( 50)  |00010                                         2  [ 5]
   '3' ( 51)  |011001                                       19  [ 6]
   '4' ( 52)  |011010                                       1a  [ 6]
   '5' ( 53)  |011011                                       1b  [ 6]
   '6' ( 54)  |011100                                       1c  [ 6]
   '7' ( 55)  |011101                                       1d  [ 6]
   '8' ( 56)  |011110                                       1e  [ 6]
   '9' ( 57)  |011111                                       1f  [ 6]
   ':' ( 58)  |1011100                                      5c  [ 7]
   ';' ( 59)  |11111011                                     fb  [ 8]
   '<' ( 60)  |11111111|1111100                           7ffc  [15]
   '=' ( 61)  |100000                                       20  [ 6]
```

Appendix C. Examples

This appendix contains examples covering integer encoding, header field representation, and the encoding of whole lists of header fields for both requests and responses, with and without Huffman coding.

C.1. Integer Representation Examples

This section shows the representation of integer values in detail (see Section 5.1).

C.1.1. Example 1: Encoding 10 Using a 5-Bit Prefix

The value 10 is to be encoded with a 5-bit prefix.

- 10 is less than 31 (2^5 - 1) and is represented using the 5-bit prefix.

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| X | X | X | 0 | 1 | 0 | 1 | 0 |   10 stored on 5 bits
+---+---+---+---+---+---+---+---+
```

C.1.2. Example 2: Encoding 1337 Using a 5-Bit Prefix

The value I=1337 is to be encoded with a 5-bit prefix.

1337 is greater than 31 (2^5 - 1).

The 5-bit prefix is filled with its max value (31).

I = 1337 - (2^5 - 1) = 1306.

I (1306) is greater than or equal to 128, so the while loop body executes:

I % 128 == 26

26 + 128 == 154

154 is encoded in 8 bits as: 10011010

I is set to 10 (1306 / 128 == 10)

I is no longer greater than or equal to 128, so the while loop terminates.

I, now 10, is encoded in 8 bits as: 00001010.

The process ends.

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| X | X | X | 1 | 1 | 1 | 1 | 1 |  Prefix = 31, I = 1306
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |  1306>=128, encode(154), I=1306/128
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |  10<128, encode(10), done
+---+---+---+---+---+---+---+---+
```

C.1.3. Example 3: Encoding 42 Starting at an Octet Boundary

The value 42 is to be encoded starting at an octet boundary. This implies that a 8-bit prefix is used.

- 42 is less than 255 (2^8 - 1) and is represented using the 8-bit prefix.

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |   42 stored on 8 bits
+---+---+---+---+---+---+---+---+
```

C.2. Header Field Representation Examples

This section shows several independent representation examples.

C.2.1. Literal Header Field with Indexing

The header field representation uses a literal name and a literal value. The header field is added to the dynamic table.

Header list to encode:

```
custom-key: custom-header
```

Hex dump of encoded data:

```
400a 6375 7374 6f6d 2d6b 6579 0d63 7573 | @.custom-key.cus
746f 6d2d 6865 6164 6572                | tom-header
```

Decoding process:

```
40                                      | == Literal indexed ==
0a                                      |   Literal name (len = 10)
6375 7374 6f6d 2d6b 6579                | custom-key
0d                                      |   Literal value (len = 13)
6375 7374 6f6d 2d68 6561 6465 72        | custom-header
                                        | -> custom-key:
                                        |   custom-header
```

Dynamic Table (after decoding):

```
[  1] (s =  55) custom-key: custom-header
      Table size:  55
```

Decoded header list:

```
custom-key: custom-header
```

C.2.2. Literal Header Field without Indexing

The header field representation uses an indexed name and a literal value. The header field is not added to the dynamic table.

Header list to encode:

```
:path: /sample/path
```

Hex dump of encoded data:

```
040c 2f73 616d 706c 652f 7061 7468      | ../sample/path
```

Decoding process:

```
04                                      | == Literal not indexed ==
                                        |   Indexed name (idx = 4)
                                        |     :path
0c                                      |   Literal value (len = 12)
2f73 616d 706c 652f 7061 7468           | /sample/path
                                        | -> :path: /sample/path
```

Dynamic table (after decoding): empty.

Decoded header list:

```
:path: /sample/path
```

C.2.3. Literal Header Field Never Indexed

The header field representation uses a literal name and a literal value. The header field is not added to the dynamic table and must use the same representation if re-encoded by an intermediary.

Header list to encode:

```
password: secret
```

Hex dump of encoded data:

```
1008 7061 7373 776f 7264 0673 6563 7265 | ..password.secre
74                                      | t
```

Decoding process:

```
10                                      | == Literal never indexed ==
08                                      |   Literal name (len = 8)
7061 7373 776f 7264                     | password
06                                      |   Literal value (len = 6)
7365 6372 6574                          | secret
                                        | -> password: secret
```

Dynamic table (after decoding): empty.

Decoded header list:

```
password: secret
```

C.2.4. Indexed Header Field

The header field representation uses an indexed header field from the static table.

Header list to encode:

```
:method: GET
```

Hex dump of encoded data:

```
82                                      | .
```

Decoding process:

```
82                                      | == Indexed - Add ==
                                        |   idx = 2
                                        | -> :method: GET
```

Dynamic table (after decoding): empty.

Decoded header list:

```
:method: GET
```

C.3. Request Examples without Huffman Coding

This section shows several consecutive header lists, corresponding to HTTP requests, on the same connection.

C.3.1. First Request

Header list to encode:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
```

Hex dump of encoded data:

```
8286 8441 0f77 7777 2e65 7861 6d70 6c65 | ...A.www.example
2e63 6f6d                               | .com
```

Decoding process:

```
82                                      | == Indexed - Add ==
                                        |   idx = 2
                                        | -> :method: GET
86                                      | == Indexed - Add ==
                                        |   idx = 6
                                        | -> :scheme: http
84                                      | == Indexed - Add ==
                                        |   idx = 4
                                        | -> :path: /
41                                      | == Literal indexed ==
                                        |   Indexed name (idx = 1)
                                        |     :authority
0f                                      |   Literal value (len = 15)
7777 772e 6578 616d 706c 652e 636f 6d   | www.example.com
                                        | -> :authority:
                                        |   www.example.com
```

Dynamic Table (after decoding):

```
[  1] (s =  57) :authority: www.example.com
      Table size:  57
```

Decoded header list:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
```

C.3.2. Second Request

Header list to encode:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
cache-control: no-cache
```

Hex dump of encoded data:

```
8286 84be 5808 6e6f 2d63 6163 6865      | ....X.no-cache
```

Decoding process:

```
82                                      | == Indexed - Add ==
                                        |   idx = 2
                                        | -> :method: GET
86                                      | == Indexed - Add ==
                                        |   idx = 6
                                        | -> :scheme: http
84                                      | == Indexed - Add ==
                                        |   idx = 4
                                        | -> :path: /
be                                      | == Indexed - Add ==
                                        |   idx = 62
                                        | -> :authority:
                                        |   www.example.com
58                                      | == Literal indexed ==
                                        |   Indexed name (idx = 24)
                                        |     cache-control
08                                      |   Literal value (len = 8)
6e6f 2d63 6163 6865                     | no-cache
                                        | -> cache-control: no-cache
```

Dynamic Table (after decoding):

```
[  1] (s =  53) cache-control: no-cache
[  2] (s =  57) :authority: www.example.com
      Table size: 110
```

Decoded header list:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
cache-control: no-cache
```

C.3.3. Third Request

Header list to encode:

```
:method: GET
:scheme: https
:path: /index.html
:authority: www.example.com
custom-key: custom-value
```

Hex dump of encoded data:

```
8287 85bf 400a 6375 7374 6f6d 2d6b 6579 | ....@.custom-key
0c63 7573 746f 6d2d 7661 6c75 65        | .custom-value
```

Decoding process:

```
82                                      | == Indexed - Add ==
                                        |   idx = 2
                                        | -> :method: GET
87                                      | == Indexed - Add ==
                                        |   idx = 7
                                        | -> :scheme: https
85                                      | == Indexed - Add ==
                                        |   idx = 5
                                        | -> :path: /index.html
bf                                      | == Indexed - Add ==
                                        |   idx = 63
                                        | -> :authority:
                                        |   www.example.com
40                                      | == Literal indexed ==
0a                                      |   Literal name (len = 10)
6375 7374 6f6d 2d6b 6579                | custom-key
0c                                      |   Literal value (len = 12)
6375 7374 6f6d 2d76 616c 7565           | custom-value
                                        | -> custom-key:
                                        |   custom-value
```

Dynamic Table (after decoding):

```
[  1] (s =  54) custom-key: custom-value
[  2] (s =  53) cache-control: no-cache
[  3] (s =  57) :authority: www.example.com
      Table size: 164
```

Decoded header list:

```
:method: GET
:scheme: https
:path: /index.html
:authority: www.example.com
custom-key: custom-value
```

C.4. Request Examples with Huffman Coding

This section shows the same examples as the previous section but uses Huffman encoding for the literal values.

C.4.1. First Request

Header list to encode:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
```

Hex dump of encoded data:

```
8286 8441 8cf1 e3c2 e5f2 3a6b a0ab 90f4 | ...A......:k....
ff                                      | .
```

Decoding process:

```
82                                      | == Indexed - Add ==
                                        |   idx = 2
                                        | -> :method: GET
86                                      | == Indexed - Add ==
                                        |   idx = 6
                                        | -> :scheme: http
84                                      | == Indexed - Add ==
                                        |   idx = 4
                                        | -> :path: /
41                                      | == Literal indexed ==
                                        |   Indexed name (idx = 1)
                                        |     :authority
8c                                      |   Literal value (len = 12)
                                        |     Huffman encoded:
f1e3 c2e5 f23a 6ba0 ab90 f4ff           | .....:k.....
                                        |     Decoded:
                                        | www.example.com
                                        | -> :authority:
                                        |   www.example.com
```

Dynamic Table (after decoding):

```
[  1] (s =  57) :authority: www.example.com
      Table size:  57
```

Decoded header list:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
```

C.4.2. Second Request

Header list to encode:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
cache-control: no-cache
```

Hex dump of encoded data:

```
8286 84be 5886 a8eb 1064 9cbf            | ....X....d..
```

Decoding process:

```
82                                       | == Indexed - Add ==
                                         |   idx = 2
                                         | -> :method: GET
86                                       | == Indexed - Add ==
                                         |   idx = 6
                                         | -> :scheme: http
84                                       | == Indexed - Add ==
                                         |   idx = 4
                                         | -> :path: /
be                                       | == Indexed - Add ==
                                         |   idx = 62
                                         | -> :authority:
                                         |   www.example.com
58                                       | == Literal indexed ==
                                         |   Indexed name (idx = 24)
                                         |     cache-control
86                                       |   Literal value (len = 6)
                                         |     Huffman encoded:
a8eb 1064 9cbf                           | ...d..
                                         |     Decoded:
                                         | no-cache
                                         | -> cache-control: no-cache
```

Dynamic Table (after decoding):

```
[  1] (s =  53) cache-control: no-cache
[  2] (s =  57) :authority: www.example.com
      Table size: 110
```

Decoded header list:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
cache-control: no-cache
```

C.4.3. Third Request

Header list to encode:

```
:method: GET
:scheme: https
:path: /index.html
:authority: www.example.com
custom-key: custom-value
```

Hex dump of encoded data:

```
8287 85bf 4088 25a8 49e9 5ba9 7d7f 8925 | ....@.%.I.[.}..%
a849 e95b b8e8 b4bf                     | .I.[....
```

Decoding process:

```
82                                      | == Indexed - Add ==
                                        |   idx = 2
                                        | -> :method: GET
87                                      | == Indexed - Add ==
                                        |   idx = 7
                                        | -> :scheme: https
85                                      | == Indexed - Add ==
                                        |   idx = 5
                                        | -> :path: /index.html
bf                                      | == Indexed - Add ==
                                        |   idx = 63
                                        | -> :authority:
                                        |   www.example.com
40                                      | == Literal indexed ==
88                                      |   Literal name (len = 8)
                                        |     Huffman encoded:
25a8 49e9 5ba9 7d7f                     | %.I.[.}.
                                        |     Decoded:
                                        | custom-key
89                                      |   Literal value (len = 9)
                                        |     Huffman encoded:
25a8 49e9 5bb8 e8b4 bf                  | %.I.[....
                                        |     Decoded:
                                        | custom-value
                                        | -> custom-key:
                                        |   custom-value
```

Dynamic Table (after decoding):

```
[  1] (s =  54) custom-key: custom-value
[  2] (s =  53) cache-control: no-cache
[  3] (s =  57) :authority: www.example.com
      Table size: 164
```

Decoded header list:

```
:method: GET
:scheme: https
:path: /index.html
:authority: www.example.com
custom-key: custom-value
```

C.5. Response Examples without Huffman Coding

This section shows several consecutive header lists, corresponding to HTTP responses, on the same connection. The HTTP/2 setting parameter SETTINGS_HEADER_TABLE_SIZE is set to the value of 256 octets, causing some evictions to occur.

C.5.1. First Response

Header list to encode:

```
:status: 302
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

Hex dump of encoded data:

```
4803 3330 3258 0770 7269 7661 7465 611d | H.302X.privatea.
4d6f 6e2c 2032 3120 4f63 7420 3230 3133 | Mon, 21 Oct 2013
2032 303a 3133 3a32 3120 474d 546e 1768 |  20:13:21 GMTn.h
7474 7073 3a2f 2f77 7777 2e65 7861 6d70 | ttps://www.examp
6c65 2e63 6f6d                          | le.com
```

Decoding process:

```
48                                      | == Literal indexed ==
                                        |   Indexed name (idx = 8)
                                        |     :status
03                                      |   Literal value (len = 3)
3330 32                                 | 302
                                        | -> :status: 302
58                                      | == Literal indexed ==
                                        |   Indexed name (idx = 24)
                                        |     cache-control
07                                      |   Literal value (len = 7)
7072 6976 6174 65                       | private
                                        | -> cache-control: private
61                                      | == Literal indexed ==
                                        |   Indexed name (idx = 33)
                                        |     date
1d                                      |   Literal value (len = 29)
4d6f 6e2c 2032 3120 4f63 7420 3230 3133 | Mon, 21 Oct 2013
2032 303a 3133 3a32 3120 474d 54        |  20:13:21 GMT
                                        | -> date: Mon, 21 Oct 2013
                                        |   20:13:21 GMT
6e                                      | == Literal indexed ==
                                        |   Indexed name (idx = 46)
                                        |     location
17                                      |   Literal value (len = 23)
6874 7470 733a 2f2f 7777 772e 6578 616d | https://www.exam
706c 652e 636f 6d                       | ple.com
                                        | -> location:
                                        |   https://www.example.com
```

Dynamic Table (after decoding):

```
[  1] (s =  63) location: https://www.example.com
[  2] (s =  65) date: Mon, 21 Oct 2013 20:13:21 GMT
[  3] (s =  52) cache-control: private
[  4] (s =  42) :status: 302
      Table size: 222
```

Decoded header list:

```
:status: 302
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

C.5.2. Second Response

The (":status", "302") header field is evicted from the dynamic table to free space to allow adding the (":status", "307") header field.

Header list to encode:

```
:status: 307
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

Hex dump of encoded data:

```
4803 3330 37c1 c0bf                     | H.307...
```

Decoding process:

```
48                                      | == Literal indexed ==
                                        |   Indexed name (idx = 8)
                                        |     :status
03                                      |   Literal value (len = 3)
3330 37                                 | 307
                                        | - evict: :status: 302
                                        | -> :status: 307
c1                                      | == Indexed - Add ==
                                        |   idx = 65
                                        | -> cache-control: private
c0                                      | == Indexed - Add ==
                                        |   idx = 64
                                        | -> date: Mon, 21 Oct 2013
                                        |   20:13:21 GMT
bf                                      | == Indexed - Add ==
                                        |   idx = 63
                                        | -> location:
                                        |   https://www.example.com
```

Dynamic Table (after decoding):

```
[  1] (s =  42) :status: 307
[  2] (s =  63) location: https://www.example.com
[  3] (s =  65) date: Mon, 21 Oct 2013 20:13:21 GMT
[  4] (s =  52) cache-control: private
      Table size: 222
```

Decoded header list:

```
:status: 307
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

C.5.3. Third Response

Several header fields are evicted from the dynamic table during the processing of this header list.

Header list to encode:

```
:status: 200
cache-control: private
date: Mon, 21 Oct 2013 20:13:22 GMT
location: https://www.example.com
content-encoding: gzip
set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU; max-age=3600; version=1
```

Hex dump of encoded data:

```
88c1 611d 4d6f 6e2c 2032 3120 4f63 7420 | ..a.Mon, 21 Oct
3230 3133 2032 303a 3133 3a32 3220 474d | 2013 20:13:22 GM
54c0 5a04 677a 6970 7738 666f 6f3d 4153 | T.Z.gzipw8foo=AS
444a 4b48 514b 425a 584f 5157 454f 5049 | DJKHQKBZXOQWEOPI
5541 5851 5745 4f49 553b 206d 6178 2d61 | UAXQWEOIU; max-a
6765 3d33 3630 303b 2076 6572 7369 6f6e | ge=3600; version
3d31                                    | =1
```

Decoding process:

```
88                                      | == Indexed - Add ==
                                        |   idx = 8
                                        | -> :status: 200
c1                                      | == Indexed - Add ==
                                        |   idx = 65
                                        | -> cache-control: private
61                                      | == Literal indexed ==
                                        |   Indexed name (idx = 33)
                                        |     date
1d                                      |   Literal value (len = 29)
4d6f 6e2c 2032 3120 4f63 7420 3230 3133 | Mon, 21 Oct 2013
2032 303a 3133 3a32 3220 474d 54        |  20:13:22 GMT
                                        | - evict: cache-control:
                                        |   private
                                        | -> date: Mon, 21 Oct 2013
                                        |   20:13:22 GMT
c0                                      | == Indexed - Add ==
                                        |   idx = 64
                                        | -> location:
                                        |   https://www.example.com
5a                                      | == Literal indexed ==
                                        |   Indexed name (idx = 26)
                                        |     content-encoding
04                                      |   Literal value (len = 4)
677a 6970                               | gzip
                                        | - evict: date: Mon, 21 Oct
                                        |    2013 20:13:21 GMT
                                        | -> content-encoding: gzip
77                                      | == Literal indexed ==
                                        |   Indexed name (idx = 55)
                                        |     set-cookie
38                                      |   Literal value (len = 56)
666f 6f3d 4153 444a 4b48 514b 425a 584f | foo=ASDJKHQKBZXO
5157 454f 5049 5541 5851 5745 4f49 553b | QWEOPIUAXQWEOIU;
206d 6178 2d61 6765 3d33 3630 303b 2076 |  max-age=3600; v
6572 7369 6f6e 3d31                     | ersion=1
                                        | - evict: location:
                                        |   https://www.example.com
                                        | - evict: :status: 307
                                        | -> set-cookie: foo=ASDJKHQ
                                        |   KBZXOQWEOPIUAXQWEOIU; ma
                                        |   x-age=3600; version=1
```

Dynamic Table (after decoding):

```
[  1] (s =  98) set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU;
                 max-age=3600; version=1
[  2] (s =  52) content-encoding: gzip
[  3] (s =  65) date: Mon, 21 Oct 2013 20:13:22 GMT
      Table size: 215
```

Decoded header list:

```
:status: 200
cache-control: private
date: Mon, 21 Oct 2013 20:13:22 GMT
location: https://www.example.com
content-encoding: gzip
set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU; max-age=3600; version=1
```

C.6. Response Examples with Huffman Coding

This section shows the same examples as the previous section but uses Huffman encoding for the literal values. The HTTP/2 setting parameter SETTINGS_HEADER_TABLE_SIZE is set to the value of 256 octets, causing some evictions to occur. The eviction mechanism uses the length of the decoded literal values, so the same evictions occur as in the previous section.

C.6.1. First Response

Header list to encode:

```
:status: 302
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

Hex dump of encoded data:

```
4882 6402 5885 aec3 771a 4b61 96d0 7abe | H.d.X...w.Ka..z.
9410 54d4 44a8 2005 9504 0b81 66e0 82a6 | ..T.D. .....f...
2d1b ff6e 919d 29ad 1718 63c7 8f0b 97c8 | -..n..)...c.....
e9ae 82ae 43d3                          | ....C.
```

Decoding process:

```
48                                      | == Literal indexed ==
                                        |   Indexed name (idx = 8)
                                        |     :status
82                                      |   Literal value (len = 2)
                                        |     Huffman encoded:
6402                                    | d.
                                        |     Decoded:
                                        | 302
                                        | -> :status: 302
58                                      | == Literal indexed ==
                                        |   Indexed name (idx = 24)
                                        |     cache-control
85                                      |   Literal value (len = 5)
                                        |     Huffman encoded:
aec3 771a 4b                            | ..w.K
                                        |     Decoded:
                                        | private
                                        | -> cache-control: private
61                                      | == Literal indexed ==
                                        |   Indexed name (idx = 33)
                                        |     date
96                                      |   Literal value (len = 22)
                                        |     Huffman encoded:
d07a be94 1054 d444 a820 0595 040b 8166 | .z...T.D. .....f
e082 a62d 1bff                          | ...-..
                                        |     Decoded:
                                        | Mon, 21 Oct 2013 20:13:21
                                        | GMT
                                        | -> date: Mon, 21 Oct 2013
                                        |   20:13:21 GMT
6e                                      | == Literal indexed ==
                                        |   Indexed name (idx = 46)
                                        |     location
91                                      |   Literal value (len = 17)
                                        |     Huffman encoded:
9d29 ad17 1863 c78f 0b97 c8e9 ae82 ae43 | .)...c.........C
d3                                      | .
                                        |     Decoded:
                                        | https://www.example.com
                                        | -> location:
                                        |   https://www.example.com
```

Dynamic Table (after decoding):

```
[  1] (s =  63) location: https://www.example.com
[  2] (s =  65) date: Mon, 21 Oct 2013 20:13:21 GMT
[  3] (s =  52) cache-control: private
[  4] (s =  42) :status: 302
      Table size: 222
```

Decoded header list:

```
:status: 302
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

C.6.2. Second Response

The (":status", "302") header field is evicted from the dynamic table to free space to allow adding the (":status", "307") header field.

Header list to encode:

```
:status: 307
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

Hex dump of encoded data:

```
4883 640e ffc1 c0bf                     | H.d.....
```

Decoding process:

```
48                                      | == Literal indexed ==
                                        |   Indexed name (idx = 8)
                                        |     :status
83                                      |   Literal value (len = 3)
                                        |     Huffman encoded:
640e ff                                 | d..
                                        |     Decoded:
                                        | 307
                                        | - evict: :status: 302
                                        | -> :status: 307
c1                                      | == Indexed - Add ==
                                        |   idx = 65
                                        | -> cache-control: private
c0                                      | == Indexed - Add ==
                                        |   idx = 64
                                        | -> date: Mon, 21 Oct 2013
                                        |   20:13:21 GMT
bf                                      | == Indexed - Add ==
                                        |   idx = 63
                                        | -> location:
                                        |   https://www.example.com
```

Dynamic Table (after decoding):

```
[  1] (s =  42) :status: 307
[  2] (s =  63) location: https://www.example.com
[  3] (s =  65) date: Mon, 21 Oct 2013 20:13:21 GMT
[  4] (s =  52) cache-control: private
      Table size: 222
```

Decoded header list:

```
:status: 307
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

C.6.3. Third Response

Several header fields are evicted from the dynamic table during the processing of this header list.

Header list to encode:

```
:status: 200
cache-control: private
date: Mon, 21 Oct 2013 20:13:22 GMT
location: https://www.example.com
content-encoding: gzip
set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU; max-age=3600; version=1
```

Hex dump of encoded data:

```
88c1 6196 d07a be94 1054 d444 a820 0595 | ..a..z...T.D. ..
040b 8166 e084 a62d 1bff c05a 839b d9ab | ...f...-...Z....
77ad 94e7 821d d7f2 e6c7 b335 dfdf cd5b | w..........5...[
3960 d5af 2708 7f36 72c1 ab27 0fb5 291f | 9`..'..6r..'..).
9587 3160 65c0 03ed 4ee5 b106 3d50 07   | ..1`e...N...=P.
```

Decoding process:

```
88                                      | == Indexed - Add ==
                                        |   idx = 8
                                        | -> :status: 200
c1                                      | == Indexed - Add ==
                                        |   idx = 65
                                        | -> cache-control: private
61                                      | == Literal indexed ==
                                        |   Indexed name (idx = 33)
                                        |     date
96                                      |   Literal value (len = 22)
                                        |     Huffman encoded:
d07a be94 1054 d444 a820 0595 040b 8166 | .z...T.D. .....f
e084 a62d 1bff                          | ...-..
                                        |     Decoded:
                                        | Mon, 21 Oct 2013 20:13:22
                                        | GMT
                                        | - evict: cache-control:
                                        |   private
                                        | -> date: Mon, 21 Oct 2013
                                        |   20:13:22 GMT
c0                                      | == Indexed - Add ==
                                        |   idx = 64
                                        | -> location:
                                        |   https://www.example.com
5a                                      | == Literal indexed ==
                                        |   Indexed name (idx = 26)
                                        |     content-encoding
83                                      |   Literal value (len = 3)
                                        |     Huffman encoded:
9bd9 ab                                 | ...
                                        |     Decoded:
                                        | gzip
                                        | - evict: date: Mon, 21 Oct
                                        |    2013 20:13:21 GMT
                                        | -> content-encoding: gzip
77                                      | == Literal indexed ==
                                        |   Indexed name (idx = 55)
                                        |     set-cookie
ad                                      |   Literal value (len = 45)
                                        |     Huffman encoded:
94e7 821d d7f2 e6c7 b335 dfdf cd5b 3960 | .........5...[9`
d5af 2708 7f36 72c1 ab27 0fb5 291f 9587 | ..'..6r..'..)...
3160 65c0 03ed 4ee5 b106 3d50 07        | 1`e...N...=P.
                                        |     Decoded:
                                        | foo=ASDJKHQKBZXOQWEOPIUAXQ
                                        | WEOIU; max-age=3600; versi
                                        | on=1
                                        | - evict: location:
                                        |   https://www.example.com
                                        | - evict: :status: 307
                                        | -> set-cookie: foo=ASDJKHQ
                                        |   KBZXOQWEOPIUAXQWEOIU; ma
                                        |   x-age=3600; version=1
```

Dynamic Table (after decoding):

```
[  1] (s =  98) set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU;
                 max-age=3600; version=1
[  2] (s =  52) content-encoding: gzip
[  3] (s =  65) date: Mon, 21 Oct 2013 20:13:22 GMT
      Table size: 215
```

Decoded header list:

```
:status: 200
cache-control: private
date: Mon, 21 Oct 2013 20:13:22 GMT
location: https://www.example.com
content-encoding: gzip
set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU; max-age=3600; version=1
```

Acknowledgments

This specification includes substantial input from the following individuals:

- Mike Bishop, Jeff Pinner, Julian Reschke, and Martin Thomson (substantial editorial contributions).
- Johnny Graettinger (Huffman code statistics).

Authors' Addresses

Roberto Peon
Google, Inc
Email: fenix@google.com

Hervé Ruellan
Canon CRF
Email: herve.ruellan@crf.canon.fr