Workgroup: QUIC
Internet-Draft: draft-ietf-quic-qpack-13
Published: 21 February 2020
Intended Status: Standards Track
Expires: 24 August 2020
Authors: C. Krasic (Netflix), M. Bishop (Akamai Technologies), A. Frindell, Ed. (Facebook)

# QPACK: Header Compression for HTTP/3

## Abstract

This specification defines QPACK, a compression format for efficiently representing HTTP header fields, to be used in HTTP/3. This is a variation of HPACK header compression that seeks to reduce head-of-line blocking.

## Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic.

Working Group information can be found at https://github.com/quicwg; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-qpack.

## 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 24 August 2020.

## Copyright Notice

**Table of Contents**

1. Introduction

1.1. Conventions and Definitions

1.2. Notational Conventions

2. Compression Process Overview

2.1. Encoder

2.1.1. Reference Tracking

2.1.2. Limits on Dynamic Table Insertions

2.1.3. Blocked Streams

2.1.4. Avoiding Flow Control Deadlocks

2.1.5. Known Received Count

2.2. Decoder

2.2.1. Blocked Decoding

2.2.2. State Synchronization

2.2.3. Invalid References

3. Header Tables

3.1. Static Table

3.2. Dynamic Table

3.2.1. Dynamic Table Size

3.2.2. Dynamic Table Capacity and Eviction

3.2.3. Maximum Dynamic Table Capacity

3.2.4. Absolute Indexing

3.2.5. Relative Indexing

3.2.6. Post-Base Indexing

4. Wire Format

4.1. Primitives

4.1.1. Prefixed Integers

4.1.2. String Literals

4.2. Encoder and Decoder Streams

4.3. Encoder Instructions

4.3.1. Set Dynamic Table Capacity

4.3.2. Insert With Name Reference

4.3.3. Insert Without Name Reference

4.3.4. Duplicate

4.4. Decoder Instructions

4.4.1. Header Acknowledgement

4.4.2. Stream Cancellation

4.4.3. Insert Count Increment

4.5. Header Block Representations

4.5.1. Header Block Prefix

4.5.2. Indexed Header Field

4.5.3. Indexed Header Field With Post-Base Index

4.5.4. Literal Header Field With Name Reference

4.5.5. Literal Header Field With Post-Base Name Reference

4.5.6. Literal Header Field Without Name Reference

5. Configuration

6. Error Handling

7. Security Considerations

8. IANA Considerations

  8.1. Settings Registration

  8.2. Stream Type Registration

  8.3. Error Code Registration

9. References

  9.1. Normative References

  9.2. Informative References

Appendix A. Static Table

Appendix B. Sample One Pass Encoding Algorithm

Appendix C. Change Log

  C.1. Since draft-ietf-quic-qpack-12

  C.2. Since draft-ietf-quic-qpack-11

  C.3. Since draft-ietf-quic-qpack-10

  C.4. Since draft-ietf-quic-qpack-09

  C.5. Since draft-ietf-quic-qpack-08

  C.6. Since draft-ietf-quic-qpack-06

  C.7. Since draft-ietf-quic-qpack-05

  C.8. Since draft-ietf-quic-qpack-04

  C.9. Since draft-ietf-quic-qpack-03

  C.10. Since draft-ietf-quic-qpack-02

  C.11. Since draft-ietf-quic-qpack-01

  C.12. Since draft-ietf-quic-qpack-00

  C.13. Since draft-ietf-quic-qcram-00

Acknowledgments

Authors' Addresses

## 1. Introduction

The QUIC transport protocol [QUIC-TRANSPORT] is designed to support HTTP semantics, and its design subsumes many of the features of HTTP/2 [RFC7540]. HTTP/2 uses HPACK [RFC7541] for header compression. If HPACK were used for HTTP/3 [HTTP3], it would induce head-of-line blocking due to built-in assumptions of a total ordering across frames on all streams.

QPACK reuses core concepts from HPACK, but is redesigned to allow correctness in the presence of out-of-order delivery, with flexibility for implementations to balance between resilience against head-of-line blocking and optimal compression ratio. The design goals are to closely approach the compression ratio of HPACK with substantially less head-of-line blocking under the same loss conditions.

### 1.1. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Definitions of terms that are used in this document:

**Header field:** A name-value pair sent as part of an HTTP message.

**Header list:** An ordered collection of header fields associated with an HTTP message. A header list can contain multiple header fields with the same name. It can also contain duplicate header fields.

**Header block:** The compressed representation of a header list.

**Encoder:** An implementation which transforms a header list into a header block.

**Decoder:** An implementation which transforms a header block into a header list.

**Absolute Index:** A unique index for each entry in the dynamic table.

**Base:** A reference point for relative and post-base indices. References to dynamic table entries in header blocks are relative to a Base.

**Insert Count:** The total number of entries inserted in the dynamic table.

QPACK is a name, not an acronym.

### 1.2. Notational Conventions

Diagrams use the format described in Section 3.1 of [RFC2360], with the following additional conventions:

**x (A)** Indicates that x is A bits long

**x (A+)** Indicates that x uses the prefixed integer encoding defined in Section 4.1.1, beginning with an A-bit prefix.

**x ...** Indicates that x is variable-length and extends to the end of the region.

## 2. Compression Process Overview

Like HPACK, QPACK uses two tables for associating header fields to indices. The static table (Section 3.1) is predefined and contains common header fields (some of them with an empty value). The dynamic table (Section 3.2) is built up over the course of the connection and can be used by the encoder to index header fields in the encoded header lists.

QPACK defines unidirectional streams for sending instructions from encoder to decoder and vice versa.

### 2.1. Encoder

An encoder converts a header list into a header block by emitting either an indexed or a literal representation for each header field in the list; see Section 4.5. Indexed representations achieve high compression by replacing the literal name and possibly the value with an index to either the static or dynamic table. References to the static table and literal representations do not require any dynamic state and never risk head-of-line blocking. References to the dynamic table risk head-of-line blocking if the encoder has not received an acknowledgement indicating the entry is available at the decoder.

An encoder MAY insert any entry in the dynamic table it chooses; it is not limited to header fields it is compressing.

QPACK preserves the ordering of header fields within each header list. An encoder MUST emit header field representations in the order they appear in the input header list.

QPACK is designed to contain the more complex state tracking to the encoder, while the decoder is relatively simple.

#### 2.1.1. Reference Tracking

An encoder MUST ensure that a header block which references a dynamic table entry is not processed by the decoder after the referenced entry has been evicted. Hence the encoder needs to retain information about each compressed header block that references the dynamic table until that header block is acknowledged by the decoder; see Section 4.4.1.

#### 2.1.2. Limits on Dynamic Table Insertions

Inserting entries into the dynamic table might not be possible if the table contains entries which cannot be evicted.

A dynamic table entry cannot be evicted immediately after insertion, even if it has never been referenced. Once the insertion of a dynamic table entry has been acknowledged and there are no outstanding unacknowledged references to the entry, the entry becomes evictable.

If the dynamic table does not contain enough room for a new entry without evicting other entries, and the entries which would be evicted are not evictable, the encoder MUST NOT insert that entry into the dynamic table (including duplicates of existing entries). In order to avoid this, an encoder that uses the dynamic table has to keep track of whether each entry is currently evictable or not.

##### 2.1.2.1. Avoiding Prohibited Insertions

To ensure that the encoder is not prevented from adding new entries, the encoder can avoid referencing entries that are close to eviction. Rather than reference such an entry, the encoder can emit a Duplicate instruction (Section 4.3.4), and reference the duplicate instead.

Determining which entries are too close to eviction to reference is an encoder preference. One heuristic is to target a fixed amount of available space in the dynamic table: either unused space or space that can be reclaimed by evicting non-blocking entries. To achieve this, the encoder can maintain a draining index, which is the smallest absolute index (Section 3.2.4) in the dynamic table that it will emit a reference for. As new entries are inserted, the encoder increases the draining index to maintain the section of the table that it will not reference. If the encoder does not create new references to entries with an absolute index lower than the draining index, the number of unacknowledged references to those entries will eventually become zero, allowing them to be evicted.

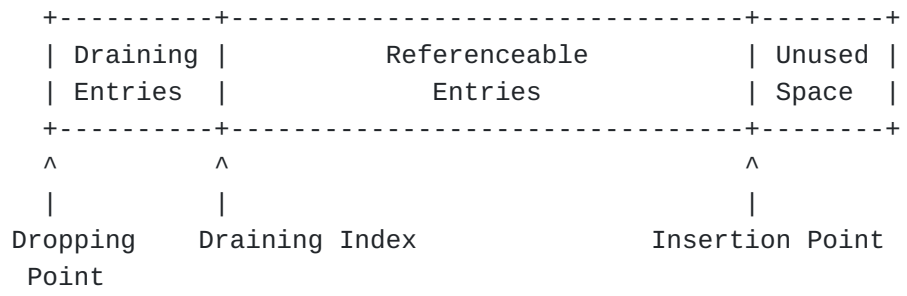



Figure 1: Draining Dynamic Table Entries

#### 2.1.3. Blocked Streams

Because QUIC does not guarantee order between data on different streams, a decoder might encounter a header block that references a dynamic table entry that it has not yet received.

Each header block contains a Required Insert Count (Section 4.5.1), the lowest possible value for the Insert Count with which the header block can be decoded. For a header block with references to the dynamic table, the Required Insert Count is one larger than the largest absolute index of all referenced dynamic table entries. For a header block with no references to the dynamic table, the Required Insert Count is zero.

When the decoder receives a header block with a Required Insert Count greater than its own Insert Count, the stream cannot be processed immediately, and is considered "blocked"; see Section 2.2.1.

The decoder specifies an upper bound on the number of streams which can be blocked using the SETTINGS_QPACK_BLOCKED_STREAMS setting; see Section 5. An encoder MUST limit the number of streams which could become blocked to the value of SETTINGS_QPACK_BLOCKED_STREAMS at all times. If a decoder encounters more blocked streams than it promised to support, it MUST treat this as a connection error of type QPACK_DECOMPRESSION_FAILED.

Note that the decoder might not become blocked on every stream which risks becoming blocked.

An encoder can decide whether to risk having a stream become blocked. If permitted by the value of SETTINGS_QPACK_BLOCKED_STREAMS, compression efficiency can often be improved by referencing dynamic table entries that are still in transit, but if there is loss or reordering the stream can become blocked at the decoder. An encoder can avoid the risk of blocking by only referencing dynamic table entries which have been acknowledged, but this could mean using literals. Since literals make the header

block larger, this can result in the encoder becoming blocked on congestion or flow control limits.

#### 2.1.4. Avoiding Flow Control Deadlocks

Writing instructions on streams that are limited by flow control can produce deadlocks.

A decoder might stop issuing flow control credit on the stream that carries a header block until the necessary updates are received on the encoder stream. If the granting of flow control credit on the encoder stream (or the connection as a whole) depends on the consumption and release of data on the stream carrying the header block, a deadlock might result.

More generally, a stream containing a large instruction can become deadlocked if the decoder withholds flow control credit until the instruction is completely received.

To avoid these deadlocks, an encoder SHOULD avoid writing an instruction unless sufficient stream and connection flow control credit is available for the entire instruction.

#### 2.1.5. Known Received Count

The Known Received Count is the total number of dynamic table insertions and duplications acknowledged by the decoder. The encoder tracks the Known Received Count in order to identify which dynamic table entries can be referenced without potentially blocking a stream. The decoder tracks the Known Received Count in order to be able to send Insert Count Increment instructions.

A Header Acknowledgement instruction (Section 4.4.1) implies that the decoder has received all dynamic table state necessary to process corresponding the header block. If the Required Insert Count of the acknowledged header block is greater than the current Known Received Count, Known Received Count is updated to the value of the Required Insert Count.

An Insert Count Increment instruction Section 4.4.3 increases the Known Received Count by its Increment parameter. See Section 2.2.2.3 for guidance.

### 2.2. Decoder

As in HPACK, the decoder processes header blocks and emits the corresponding header lists. It also processes instructions received on the encoder stream that modify the dynamic table. Note that header blocks and encoder stream instructions arrive on separate streams. This is unlike HPACK, where header blocks can contain

instructions that modify the dynamic table, and there is no dedicated stream of HPACK instructions.

The decoder MUST emit header fields in the order their representations appear in the input header block.

#### 2.2.1. Blocked Decoding

Upon receipt of a header block, the decoder examines the Required Insert Count. When the Required Insert Count is less than or equal to the decoder's Insert Count, the header block can be processed immediately. Otherwise, the stream on which the header block was received becomes blocked.

While blocked, header block data SHOULD remain in the blocked stream's flow control window. A stream becomes unblocked when the Insert Count becomes greater than or equal to the Required Insert Count for all header blocks the decoder has started reading from the stream.

When processing header blocks, the decoder expects the Required Insert Count to exactly match the value defined in Section 2.1.3. If it encounters a smaller value than expected, it MUST treat this as a connection error of type QPACK_DECOMPRESSION_FAILED; see Section 2.2.3. If it encounters a larger value than expected, it MAY treat this as a connection error of type QPACK_DECOMPRESSION_FAILED.

#### 2.2.2. State Synchronization

The decoder signals the following events by emitting decoder instructions (Section 4.4) on the decoder stream.

##### 2.2.2.1. Completed Processing of a Header Block

After the decoder finishes decoding a header block containing dynamic table references, it MUST emit a Header Acknowledgement instruction (Section 4.4.1). A stream may carry multiple header blocks in the case of intermediate responses, trailers, and pushed requests. The encoder interprets each Header Acknowledgement instruction as acknowledging the earliest unacknowledged header block containing dynamic table references sent on the given stream.

##### 2.2.2.2. Abandonment of a Stream

When an endpoint receives a stream reset before the end of a stream or before all header blocks are processed on that stream, or when it abandons reading of a stream, it generates a Stream Cancellation instruction; see Section 4.4.2. This signals to the encoder that all references to the dynamic table on that stream are no longer outstanding. A decoder with a maximum dynamic table capacity

(Section 3.2.3) equal to zero MAY omit sending Stream Cancellations, because the encoder cannot have any dynamic table references. An encoder cannot infer from this instruction that any updates to the dynamic table have been received.

The Header Acknowledgement and Stream Cancellation instructions permit the encoder to remove references to entries in the dynamic table. When an entry with absolute index lower than the Known Received Count has zero references, then it is considered evictable; see Section 2.1.2.

#### 2.2.2.3. New Table Entries

After receiving new table entries on the encoder stream, the decoder chooses when to emit Insert Count Increment instructions; see Section 4.4.3. Emitting this instruction after adding each new dynamic table entry will provide the timeliest feedback to the encoder, but could be redundant with other decoder feedback. By delaying an Insert Count Increment instruction, the decoder might be able to coalesce multiple Insert Count Increment instructions, or replace them entirely with Header Acknowledgements; see Section 4.4.1. However, delaying too long may lead to compression inefficiencies if the encoder waits for an entry to be acknowledged before using it.

### 2.2.3. Invalid References

If the decoder encounters a reference in a header block representation to a dynamic table entry which has already been evicted or which has an absolute index greater than or equal to the declared Required Insert Count (Section 4.5.1), it MUST treat this as a connection error of type QPACK_DECOMPRESSION_FAILED.

If the decoder encounters a reference in an encoder instruction to a dynamic table entry which has already been evicted, it MUST treat this as a connection error of type QPACK_ENCODER_STREAM_ERROR.

## 3. Header Tables

Unlike in HPACK, entries in the QPACK static and dynamic tables are addressed separately. The following sections describe how entries in each table are addressed.

### 3.1. Static Table

The static table consists of a predefined static list of header fields, each of which has a fixed index over time. Its entries are defined in Appendix A.

All entries in the static table have a name and a value. However, values can be empty (that is, have a length of 0). Each entry is identified by a unique index.

Note that the QPACK static table is indexed from 0, whereas the HPACK static table is indexed from 1.

When the decoder encounters an invalid static table index in a header block representation it MUST treat this as a connection error of type QPACK_DECOMPRESSION_FAILED. If this index is received on the encoder stream, this MUST be treated as a connection error of type QPACK_ENCODER_STREAM_ERROR.

### 3.2. Dynamic Table

The dynamic table consists of a list of header fields maintained in first-in, first-out order. Each HTTP/3 endpoint holds a dynamic table that is initially empty. Entries are added by encoder instructions received on the encoder stream; see Section 4.3.

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 the decoder.

Dynamic table entries can have empty values.

#### 3.2.1. Dynamic 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 bytes, its value's length in bytes, and 32. The size of an entry is calculated using the length of its name and value without Huffman encoding applied.

#### 3.2.2. Dynamic Table Capacity and Eviction

The encoder sets the capacity of the dynamic table, which serves as the upper limit on its size. The initial capacity of the dynamic table is zero. The encoder sends a Set Dynamic Table Capacity instruction (Section 4.3.1) with a non-zero capacity to begin using the dynamic table.

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 (table capacity - size of new entry). The encoder MUST NOT cause a dynamic table entry to be evicted unless that entry is evictable; see Section 2.1.2. The new entry is then added to the table. It is an error if the encoder attempts to add an entry that is larger than the dynamic table

capacity; the decoder MUST treat this as a connection error of type QPACK_ENCODER_STREAM_ERROR.

A new entry can reference 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 or value if the referenced entry is evicted from the dynamic table prior to inserting the new entry.

Whenever the dynamic table capacity is reduced by the encoder (Section 4.3.1), entries are evicted from the end of the dynamic table until the size of the dynamic table is less than or equal to the new table capacity. This mechanism can be used to completely clear entries from the dynamic table by setting a capacity of 0, which can subsequently be restored.

#### 3.2.3. Maximum Dynamic Table Capacity

To bound the memory requirements of the decoder, the decoder limits the maximum value the encoder is permitted to set for the dynamic table capacity. In HTTP/3, this limit is determined by the value of SETTINGS_QPACK_MAX_TABLE_CAPACITY sent by the decoder; see Section 5. The encoder MUST not set a dynamic table capacity that exceeds this maximum, but it can choose to use a lower dynamic table capacity; see Section 4.3.1.

For clients using 0-RTT data in HTTP/3, the server's maximum table capacity is the remembered value of the setting, or zero if the value was not previously sent. When the client's 0-RTT value of the SETTING is zero, the server MAY set it to a non-zero value in its SETTINGS frame. If the remembered value is non-zero, the server MUST send the same non-zero value in its SETTINGS frame. If it specifies any other value, or omits SETTINGS_QPACK_MAX_TABLE_CAPACITY from SETTINGS, the encoder must treat this as a connection error of type QPACK_DECODER_STREAM_ERROR.

For HTTP/3 servers and HTTP/3 clients when 0-RTT is not attempted or is rejected, the maximum table capacity is 0 until the encoder processes a SETTINGS frame with a non-zero value of SETTINGS_QPACK_MAX_TABLE_CAPACITY.

When the maximum table capacity is zero, the encoder MUST NOT insert entries into the dynamic table, and MUST NOT send any encoder instructions on the encoder stream.

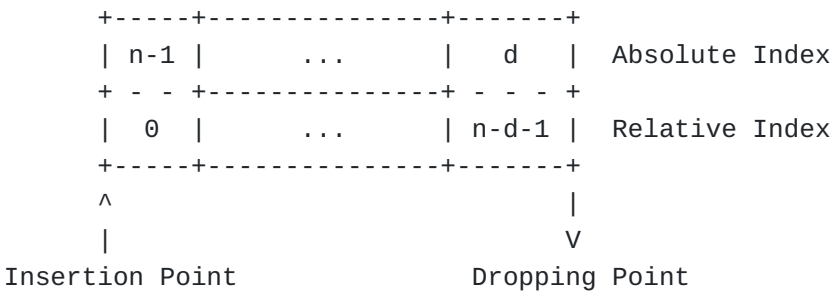
#### 3.2.4. Absolute Indexing

Each entry possesses an absolute index which is fixed for the lifetime of that entry. The first entry inserted has an absolute index of "0"; indices increase by one with each insertion.

### 3.2.5. Relative Indexing

Relative indices begin at zero and increase in the opposite direction from the absolute index. Determining which entry has a relative index of "0" depends on the context of the reference.

In encoder instructions (Section 4.3), a relative index of "0" refers to the most recently inserted value in the dynamic table. Note that this means the entry referenced by a given relative index will change while interpreting instructions on the encoder stream.

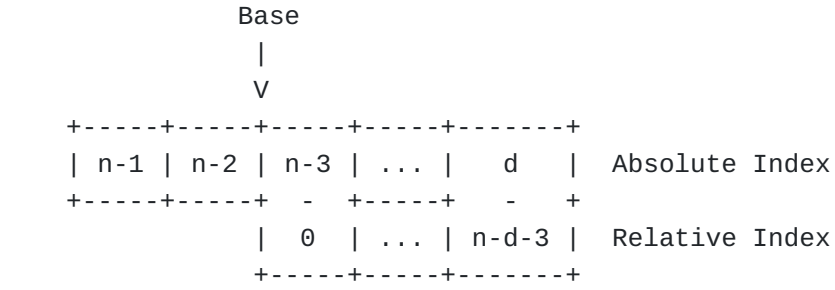



n = count of entries inserted
d = count of entries dropped

Figure 2: Example Dynamic Table Indexing - Encoder Stream

Unlike in encoder instructions, relative indices in header block representations are relative to the Base at the beginning of the header block; see Section 4.5.1. This ensures that references are stable even if header blocks and dynamic table updates are processed out of order.

In a header block a relative index of "0" refers to the entry with absolute index equal to Base - 1.



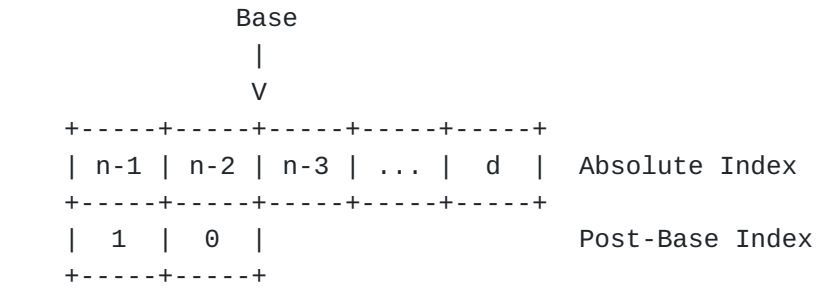

n = count of entries inserted
d = count of entries dropped
In this example, Base = n - 2

Figure 3: Example Dynamic Table Indexing - Relative Index in Header Block

### 3.2.6. Post-Base Indexing

Post-Base indices are used in header block instructions for entries with absolute indices greater than or equal to Base, starting at 0 for the entry with absolute index equal to Base, and increasing in the same direction as the absolute index.

Post-Base indices allow an encoder to process a header block in a single pass and include references to entries added while processing this (or other) header blocks.




n = count of entries inserted
d = count of entries dropped
In this example, Base = n - 2

Figure 4: Example Dynamic Table Indexing - Post-Base Index in Header Block

## 4. Wire Format

### 4.1. Primitives

#### 4.1.1. Prefixed Integers

The prefixed integer from Section 5.1 of [RFC7541] is used heavily throughout this document. The format from [RFC7541] is used unmodified. Note, however, that QPACK uses some prefix sizes not actually used in HPACK.

QPACK implementations MUST be able to decode integers up to and including 62 bits long.

#### 4.1.2. String Literals

The string literal defined by Section 5.2 of [RFC7541] is also used throughout. This string format includes optional Huffman encoding.

HPACK defines string literals to begin on a byte boundary. They begin with a single bit flag, denoted as 'H' in this document (indicating whether the string is Huffman-coded), followed by the Length encoded as a 7-bit prefix integer, and finally Length bytes of data. When Huffman encoding is enabled, the Huffman table from Appendix B of [RFC7541] is used without modification.

This document expands the definition of string literals and permits them to begin other than on a byte boundary. An "N-bit prefix string literal" begins with the same Huffman flag, followed by the length encoded as an (N-1)-bit prefix integer. The prefix size, N, can have a value between 2 and 8 inclusive. The remainder of the string literal is unmodified.

A string literal without a prefix length noted is an 8-bit prefix string literal and follows the definitions in [RFC7541] without modification.

### 4.2. Encoder and Decoder Streams

QPACK defines two unidirectional stream types:

* An encoder stream is a unidirectional stream of type 0x02. It carries an unframed sequence of encoder instructions from encoder to decoder.

* A decoder stream is a unidirectional stream of type 0x03. It carries an unframed sequence of decoder instructions from decoder to encoder.

HTTP/3 endpoints contain a QPACK encoder and decoder. Each endpoint MUST initiate at most one encoder stream and at most one decoder stream. Receipt of a second instance of either stream type MUST be treated as a connection error of type HTTP_STREAM_CREATION_ERROR. These streams MUST NOT be closed. Closure of either unidirectional stream type MUST be treated as a connection error of type HTTP_CLOSED_CRITICAL_STREAM.

An endpoint MAY avoid creating an encoder stream if it's not going to be used (for example if its encoder doesn't wish to use the dynamic table, or if the maximum size of the dynamic table permitted by the peer is zero).

An endpoint MAY avoid creating a decoder stream if its decoder sets the maximum capacity of the dynamic table to zero.

An endpoint MUST allow its peer to create an encoder stream and a decoder stream even if the connection's settings prevent their use.

### 4.3. Encoder Instructions

An encoder sends encoder instructions on the encoder stream to set the capacity of the dynamic table and add dynamic table entries. Instructions adding table entries can use existing entries to avoid transmitting redundant information. The name can be transmitted as a reference to an existing entry in the static or the dynamic table or as a string literal. For entries which already exist in the dynamic table, the full entry can also be used by reference, creating a duplicate entry.

This section specifies the following encoder instructions.

#### 4.3.1. Set Dynamic Table Capacity

An encoder informs the decoder of a change to the dynamic table capacity using an instruction which begins with the '001' three-bit pattern. This is followed by the new dynamic table capacity represented as an integer with a 5-bit prefix; see Section 4.1.1.

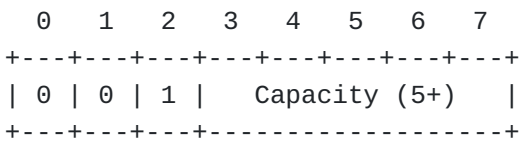



Figure 5: Set Dynamic Table Capacity

The new capacity MUST be lower than or equal to the limit described in Section 3.2.3. In HTTP/3, this limit is the value of the SETTINGS_QPACK_MAX_TABLE_CAPACITY parameter (Section 5) received from the decoder. The decoder MUST treat a new dynamic table capacity value that exceeds this limit as a connection error of type QPACK_ENCODER_STREAM_ERROR.

Reducing the dynamic table capacity can cause entries to be evicted; see Section 3.2.2. This MUST NOT cause the eviction of entries which are not evictable; see Section 2.1.2. Changing the capacity of the dynamic table is not acknowledged as this instruction does not insert an entry.

#### 4.3.2. Insert With Name Reference

An encoder adds an entry to the dynamic table where the header field name matches the header field name of an entry stored in the static or the dynamic table using an instruction that starts with the '1' one-bit pattern. The second ('T') bit indicates whether the reference is to the static or dynamic table. The 6-bit prefix integer (Section 4.1.1) that follows is used to locate the table entry for the header name. When T=1, the number represents the

static table index; when T=0, the number is the relative index of the entry in the dynamic table.

The header name reference is followed by the header field value represented as a string literal; see Section 4.1.2.

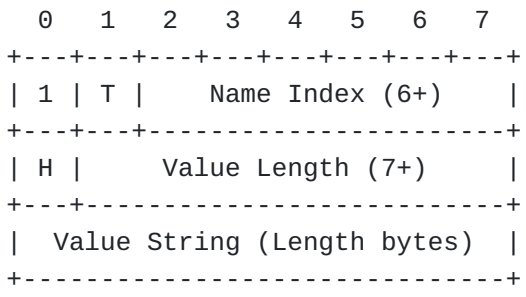



Figure 6: Insert Header Field -- Indexed Name

### 4.3.3. Insert Without Name Reference

An encoder adds an entry to the dynamic table where both the header field name and the header field value are represented as string literals using an instruction that starts with the '01' two-bit pattern.

This is followed by the name represented as a 6-bit prefix string literal, and the value represented as an 8-bit prefix string literal; see Section 4.1.2.

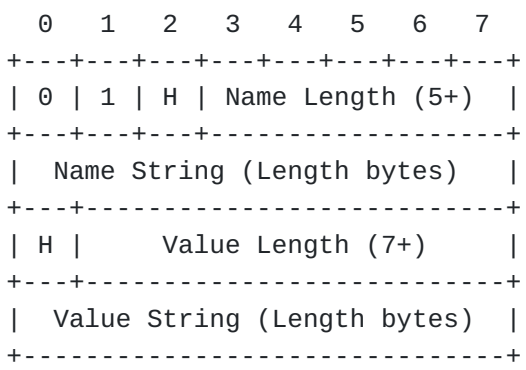



Figure 7: Insert Header Field -- New Name

### 4.3.4. Duplicate

An encoder duplicates an existing entry in the dynamic table using an instruction that begins with the '000' three-bit pattern. This is followed by the relative index of the existing entry represented as an integer with a 5-bit prefix; see Section 4.1.1.

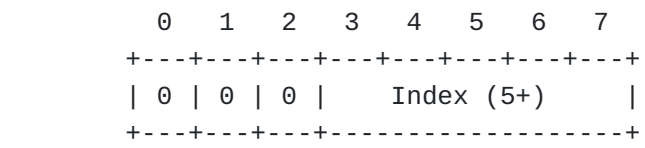



Figure 8: Duplicate

The existing entry is re-inserted into the dynamic table without resending either the name or the value. This is useful to avoid adding a reference to an older entry, which might block inserting new entries.

### 4.4. Decoder Instructions

A decoder sends decoder instructions on the decoder stream to inform the encoder about the processing of header blocks and table updates to ensure consistency of the dynamic table.

This section specifies the following decoder instructions.

#### 4.4.1. Header Acknowledgement

After processing a header block whose declared Required Insert Count is not zero, the decoder emits a Header Acknowledgement instruction. The instruction begins with the '1' one-bit pattern which is followed by the header block's associated stream ID encoded as a 7-bit prefix integer; see Section 4.1.1.

This instruction is used as described in Section 2.1.5 and in Section 2.2.2.

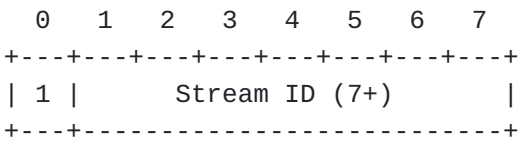



Figure 9: Header Acknowledgement

If an encoder receives a Header Acknowledgement instruction referring to a stream on which every header block with a non-zero Required Insert Count has already been acknowledged, that MUST be treated as a connection error of type QPACK_DECODER_STREAM_ERROR.

The Header Acknowledgement instruction might increase the Known Received Count; see Section 2.1.5.

### 4.4.2. Stream Cancellation

When a stream is reset or reading is abandoned, the decoder emits a Stream Cancellation instruction. The instruction begins with the '01' two-bit pattern, which is followed by the stream ID of the affected stream encoded as a 6-bit prefix integer.

This instruction is used as described in Section 2.2.2.

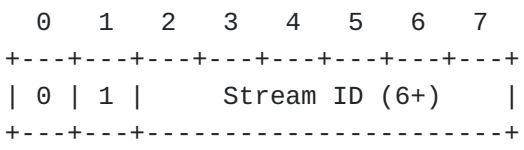



Figure 10: Stream Cancellation

### 4.4.3. Insert Count Increment

The Insert Count Increment instruction begins with the '00' two-bit pattern, followed by the Increment encoded as a 6-bit prefix integer. This instruction increases the Known Received Count (Section 2.1.5) by the value of the Increment parameter. The decoder should send an Increment value that increases the Known Received Count to the total number of dynamic table insertions and duplications processed so far.

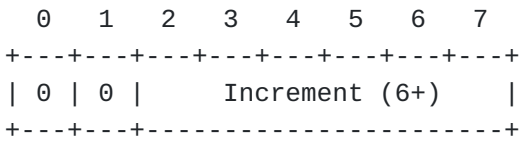



Figure 11: Insert Count Increment

An encoder that receives an Increment field equal to zero, or one that increases the Known Received Count beyond what the encoder has sent MUST treat this as a connection error of type QPACK_DECODER_STREAM_ERROR.

## 4.5. Header Block Representations

A header block consists of a prefix and a possibly empty sequence of representations defined in this section. Each representation corresponds to a single header field. These representations reference the static table or the dynamic table in a particular state, but do not modify that state.

Header blocks are carried in frames on streams defined by the enclosing protocol.

#### 4.5.1. Header Block Prefix

Each header block is prefixed with two integers. The Required Insert Count is encoded as an integer with an 8-bit prefix after the encoding described in Section 4.5.1.1). The Base is encoded as a sign bit ('S') and a Delta Base value with a 7-bit prefix; see Section 4.5.1.2.

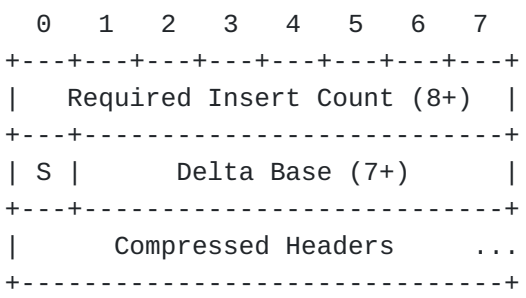



Figure 12: Header Block

##### 4.5.1.1. Required Insert Count

Required Insert Count identifies the state of the dynamic table needed to process the header block. Blocking decoders use the Required Insert Count to determine when it is safe to process the rest of the block.

The encoder transforms the Required Insert Count as follows before encoding:

```
if ReqInsertCount == 0:
   EncInsertCount = 0
else:
   EncInsertCount = (ReqInsertCount mod (2 * MaxEntries)) + 1
```

Here MaxEntries is the maximum number of entries that the dynamic table can have. The smallest entry has empty name and value strings and has the size of 32. Hence MaxEntries is calculated as

```
MaxEntries = floor( MaxTableCapacity / 32 )
```

MaxTableCapacity is the maximum capacity of the dynamic table as specified by the decoder; see Section 3.2.3.

This encoding limits the length of the prefix on long-lived connections.

The decoder can reconstruct the Required Insert Count using an algorithm such as the following. If the decoder encounters a value of EncodedInsertCount that could not have been produced by a

conformant encoder, it MUST treat this as a connection error of type QPACK_DECOMPRESSION_FAILED.

TotalNumberOfInserts is the total number of inserts into the decoder's dynamic table.

```
FullRange = 2 * MaxEntries
if EncodedInsertCount == 0:
   ReqInsertCount = 0
else:
   if EncodedInsertCount > FullRange:
      Error
   MaxValue = TotalNumberOfInserts + MaxEntries

   # MaxWrapped is the largest possible value of
   # ReqInsertCount that is 0 mod 2*MaxEntries
   MaxWrapped = floor(MaxValue / FullRange) * FullRange
   ReqInsertCount = MaxWrapped + EncodedInsertCount - 1

   # If ReqInsertCount exceeds MaxValue, the Encoder's value
   # must have wrapped one fewer time
   if ReqInsertCount > MaxValue:
      if ReqInsertCount <= FullRange:
         Error
      ReqInsertCount -= FullRange

   # Value of 0 must be encoded as 0.
   if ReqInsertCount == 0:
      Error
```

For example, if the dynamic table is 100 bytes, then the Required Insert Count will be encoded modulo 6. If a decoder has received 10 inserts, then an encoded value of 3 indicates that the Required Insert Count is 9 for the header block.

#### 4.5.1.2. Base

The Base is used to resolve references in the dynamic table as described in Section 3.2.5.

To save space, the Base is encoded relative to the Required Insert Count using a one-bit sign ('S') and the Delta Base value. A sign bit of 0 indicates that the Base is greater than or equal to the value of the Required Insert Count; the decoder adds the value of Delta Base to the Required Insert Count to determine the value of the Base. A sign bit of 1 indicates that the Base is less than the Required Insert Count; the decoder subtracts the value of Delta Base from the Required Insert Count and also subtracts one to determine the value of the Base. That is:

```
if S == 0:
   Base = ReqInsertCount + DeltaBase
else:
   Base = ReqInsertCount - DeltaBase - 1
```

A single-pass encoder determines the Base before encoding a header block. If the encoder inserted entries in the dynamic table while encoding the header block, Required Insert Count will be greater than the Base, so the encoded difference is negative and the sign bit is set to 1. If the header block did not reference the most recent entry in the table and did not insert any new entries, the Base will be greater than the Required Insert Count, so the delta will be positive and the sign bit is set to 0.

An encoder that produces table updates before encoding a header block might set Base to the value of Required Insert Count. In such case, both the sign bit and the Delta Base will be set to zero.

A header block that does not reference the dynamic table can use any value for the Base; setting Delta Base to zero is one of the most efficient encodings.

For example, with a Required Insert Count of 9, a decoder receives an S bit of 1 and a Delta Base of 2. This sets the Base to 6 and enables post-base indexing for three entries. In this example, a relative index of 1 refers to the 5th entry that was added to the table; a post-base index of 1 refers to the 8th entry.

### 4.5.2. Indexed Header Field

An indexed header field representation identifies an entry in the static table, or an entry in the dynamic table with an absolute index less than the value of the Base.

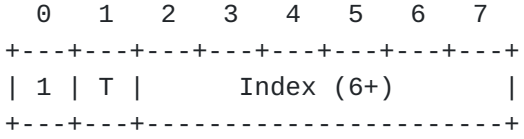



Figure 13: Indexed Header Field

This representation starts with the '1' 1-bit pattern, followed by the 'T' bit indicating whether the reference is into the static or dynamic table. The 6-bit prefix integer (Section 4.1.1) that follows is used to locate the table entry for the header field. When T=1, the number represents the static table index; when T=0, the number is the relative index of the entry in the dynamic table.

### 4.5.3. Indexed Header Field With Post-Base Index

An indexed header field with post-base index representation identifies an entry in the dynamic table with an absolute index greater than or equal to the value of the Base.

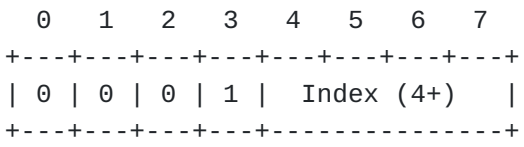



Figure 14: Indexed Header Field with Post-Base Index

This representation starts with the '0001' 4-bit pattern. This is followed by the post-base index (Section 3.2.6) of the matching header field, represented as an integer with a 4-bit prefix; see Section 4.1.1.

### 4.5.4. Literal Header Field With Name Reference

A literal header field with name reference representation encodes a header field where the header field name matches the header field name of an entry in the static table, or the header field name of an entry in the dynamic table with an absolute index less than the value of the Base.

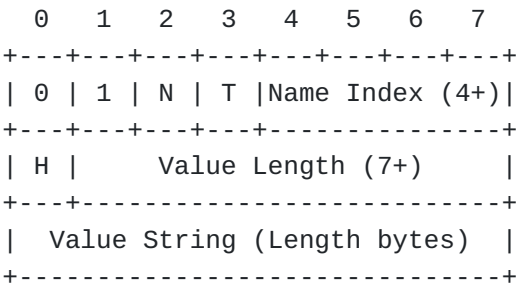



Figure 15: Literal Header Field With Name Reference

This representation starts with the '01' two-bit pattern. The following bit, 'N', indicates whether an intermediary is permitted to add this header to the dynamic header table on subsequent hops. When the 'N' bit is set, the encoded header MUST always be encoded with a literal representation. In particular, when a peer sends a header field that it received represented as a literal header field with the 'N' bit set, it MUST use a literal representation to forward this header field. This bit is intended for protecting header field values that are not to be put at risk by compressing them; see Section 7 for more details.

The fourth ('T') bit indicates whether the reference is to the static or dynamic table. The 4-bit prefix integer (Section 4.1.1) that follows is used to locate the table entry for the header name. When T=1, the number represents the static table index; when T=0, the number is the relative index of the entry in the dynamic table.

Only the header field name is taken from the dynamic table entry; the header field value is encoded as an 8-bit prefix string literal; see Section 4.1.2.

### 4.5.5. Literal Header Field With Post-Base Name Reference

A literal header field with post-base name reference representation encodes a header field where the header field name matches the header field name of a dynamic table entry with an absolute index greater than or equal to the value of the Base.

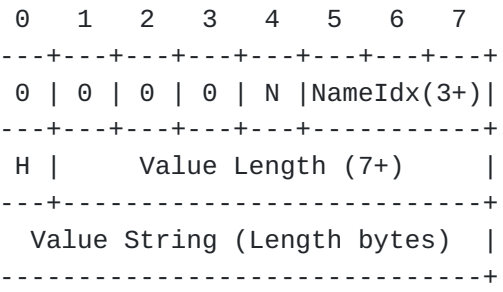



Figure 16: Literal Header Field With Post-Base Name Reference

This representation starts with the '0000' four-bit pattern. The fifth bit is the 'N' bit as described in Section 4.5.4. This is followed by a post-base index of the dynamic table entry (Section 3.2.6) encoded as an integer with a 3-bit prefix; see Section 4.1.1.

Only the header field name is taken from the dynamic table entry; the header field value is encoded as an 8-bit prefix string literal; see Section 4.1.2.

### 4.5.6. Literal Header Field Without Name Reference

The literal header field without name reference representation encodes a header field name and a header field value as string literals.

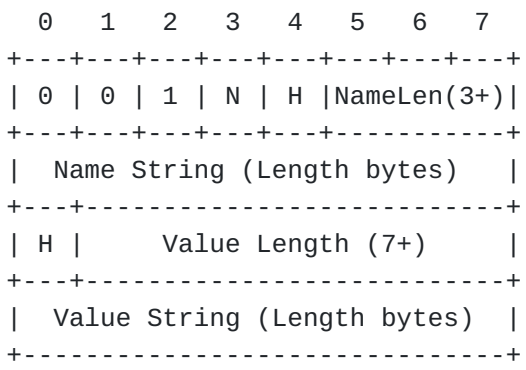



Figure 17: Literal Header Field Without Name Reference

This representation begins with the '001' three-bit pattern. The fourth bit is the 'N' bit as described in Section 4.5.4. The name follows, represented as a 4-bit prefix string literal, then the value, represented as an 8-bit prefix string literal; see Section 4.1.2.

## 5. Configuration

QPACK defines two settings which are included in the HTTP/3 SETTINGS frame.

**SETTINGS_QPACK_MAX_TABLE_CAPACITY (0x1):** The default value is zero. See Section 3.2 for usage. This is the equivalent of the SETTINGS_HEADER_TABLE_SIZE from HTTP/2.

**SETTINGS_QPACK_BLOCKED_STREAMS (0x7):** The default value is zero. See Section 2.1.3.

## 6. Error Handling

The following error codes are defined for HTTP/3 to indicate failures of QPACK which prevent the connection from continuing:

**QPACK_DECOMPRESSION_FAILED (0x200):** The decoder failed to interpret a header block and is not able to continue decoding that header block.

**QPACK_ENCODER_STREAM_ERROR (0x201):** The decoder failed to interpret an encoder instruction received on the encoder stream.

**QPACK_DECODER_STREAM_ERROR (0x202):** The encoder failed to interpret a decoder instruction received on the decoder stream.

## 7. Security Considerations

TBD. Also see Section 7.1 of [RFC7541].

While the negotiated limit on the dynamic table size accounts for much of the memory that can be consumed by a QPACK implementation, data which cannot be immediately sent due to flow control is not affected by this limit. Implementations should limit the size of unsent data, especially on the decoder stream where flexibility to choose what to send is limited. Possible responses to an excess of unsent data might include limiting the ability of the peer to open new streams, reading only from the encoder stream, or closing the connection.

## 8. IANA Considerations

### 8.1. Settings Registration

This document specifies two settings. The entries in the following table are registered in the "HTTP/3 Settings" registry established in [HTTP3].

| Setting Name | Code | Specification | Default |
|---|---|---|---|
| QPACK_MAX_TABLE_CAPACITY | 0x1 | Section 5 | 0 |
| QPACK_BLOCKED_STREAMS | 0x7 | Section 5 | 0 |

Table 1

### 8.2. Stream Type Registration

This document specifies two stream types. The entries in the following table are registered in the "HTTP/3 Stream Type" registry established in [HTTP3].

| Stream Type | Code | Specification | Sender |
|---|---|---|---|
| QPACK Encoder Stream | 0x02 | Section 4.2 | Both |
| QPACK Decoder Stream | 0x03 | Section 4.2 | Both |

Table 2

### 8.3. Error Code Registration

This document specifies three error codes. The entries in the following table are registered in the "HTTP/3 Error Code" registry established in [HTTP3].

| Name | Code | Description | Specification |
|---|---|---|---|
| QPACK_DECOMPRESSION_FAILED | 0x200 | Decompression of a header block failed | Section 6 |
| QPACK_ENCODER_STREAM_ERROR | 0x201 | Error on the encoder stream | Section 6 |
| QPACK_DECODER_STREAM_ERROR | 0x202 | Error on the decoder stream | Section 6 |

Table 3

## 9. References

### 9.1. Normative References


**[HTTP3]** Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-26, 21 February 2020, <https://tools.ietf.org/html/draft-ietf-quic-http-26>.

**[QUIC-TRANSPORT]** Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-26, 21 February 2020, <https://tools.ietf.org/html/draft-ietf-quic-transport-26>.

**[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>.

**[RFC7541]** Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

**[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>.


### 9.2. Informative References


**[RFC2360]** Scott, G., "Guide for Internet Standards Writers", BCP 22, RFC 2360, DOI 10.17487/RFC2360, June 1998, <https://www.rfc-editor.org/info/rfc2360>.

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


## Appendix A. Static Table

| Index | Name | Value |
|---|---|---|
| 0 | :authority | |
| 1 | :path | / |
| 2 | age | 0 |
| 3 | content-disposition | |
| 4 | content-length | 0 |
| 5 | cookie | |
| 6 | date | |

| Index | Name | Value |
|---|---|---|
| 7 | etag | |
| 8 | if-modified-since | |
| 9 | if-none-match | |
| 10 | last-modified | |
| 11 | link | |
| 12 | location | |
| 13 | referer | |
| 14 | set-cookie | |
| 15 | :method | CONNECT |
| 16 | :method | DELETE |
| 17 | :method | GET |
| 18 | :method | HEAD |
| 19 | :method | OPTIONS |
| 20 | :method | POST |
| 21 | :method | PUT |
| 22 | :scheme | http |
| 23 | :scheme | https |
| 24 | :status | 103 |
| 25 | :status | 200 |
| 26 | :status | 304 |
| 27 | :status | 404 |
| 28 | :status | 503 |
| 29 | accept | */* |
| 30 | accept | application/dns-message |
| 31 | accept-encoding | gzip, deflate, br |
| 32 | accept-ranges | bytes |
| 33 | access-control-allow-headers | cache-control |
| 34 | access-control-allow-headers | content-type |
| 35 | access-control-allow-origin | * |
| 36 | cache-control | max-age=0 |
| 37 | cache-control | max-age=2592000 |
| 38 | cache-control | max-age=604800 |
| 39 | cache-control | no-cache |
| 40 | cache-control | no-store |
| 41 | cache-control | public, max-age=31536000 |
| 42 | content-encoding | br |
| 43 | content-encoding | gzip |
| 44 | content-type | application/dns-message |
| 45 | content-type | application/javascript |
| 46 | content-type | application/json |
| 47 | content-type | application/x-www-form-urlencoded |
| 48 | content-type | image/gif |

| Index | Name | Value |
|---|---|---|
| 49 | content-type | image/jpeg |
| 50 | content-type | image/png |
| 51 | content-type | text/css |
| 52 | content-type | text/html; charset=utf-8 |
| 53 | content-type | text/plain |
| 54 | content-type | text/plain;charset=utf-8 |
| 55 | range | bytes=0- |
| 56 | strict-transport-security | max-age=31536000 |
| 57 | strict-transport-security | max-age=31536000; includesubdomains |
| 58 | strict-transport-security | max-age=31536000; includesubdomains; preload |
| 59 | vary | accept-encoding |
| 60 | vary | origin |
| 61 | x-content-type-options | nosniff |
| 62 | x-xss-protection | 1; mode=block |
| 63 | :status | 100 |
| 64 | :status | 204 |
| 65 | :status | 206 |
| 66 | :status | 302 |
| 67 | :status | 400 |
| 68 | :status | 403 |
| 69 | :status | 421 |
| 70 | :status | 425 |
| 71 | :status | 500 |
| 72 | accept-language | |
| 73 | access-control-allow-credentials | FALSE |
| 74 | access-control-allow-credentials | TRUE |
| 75 | access-control-allow-headers | * |
| 76 | access-control-allow-methods | get |
| 77 | access-control-allow-methods | get, post, options |
| 78 | access-control-allow-methods | options |
| 79 | access-control-expose-headers | content-length |
| 80 | access-control-request-headers | content-type |
| 81 | access-control-request-method | get |
| 82 | | post |

| Index | Name | Value |
|---|---|---|
| | access-control-request-method | |
| 83 | alt-svc | clear |
| 84 | authorization | |
| 85 | content-security-policy | script-src 'none'; object-src 'none'; base-uri 'none' |
| 86 | early-data | 1 |
| 87 | expect-ct | |
| 88 | forwarded | |
| 89 | if-range | |
| 90 | origin | |
| 91 | purpose | prefetch |
| 92 | server | |
| 93 | timing-allow-origin | * |
| 94 | upgrade-insecure-requests | 1 |
| 95 | user-agent | |
| 96 | x-forwarded-for | |
| 97 | x-frame-options | deny |
| 98 | x-frame-options | sameorigin |

Table 4

## Appendix B. Sample One Pass Encoding Algorithm

Pseudo-code for single pass encoding, excluding handling of duplicates, non-blocking mode, and reference tracking.

```
baseIndex = dynamicTable.baseIndex
largestReference = 0
for header in headers:
  staticIdx = staticTable.getIndex(header)
  if staticIdx:
    encodeIndexReference(streamBuffer, staticIdx)
    continue

  dynamicIdx = dynamicTable.getIndex(header)
  if !dynamicIdx:
    # No matching entry.  Either insert+index or encode literal
    nameIdx = getNameIndex(header)
    if shouldIndex(header) and dynamicTable.canIndex(header):
      encodeLiteralWithIncrementalIndex(controlBuffer, nameIdx,
                                        header)
      dynamicTable.add(header)
      dynamicIdx = dynamicTable.baseIndex

  if !dynamicIdx:
    # Couldn't index it, literal
    if nameIdx <= staticTable.size:
      encodeLiteral(streamBuffer, nameIndex, header)
    else:
      # encode literal, possibly with nameIdx above baseIndex
      encodeDynamicLiteral(streamBuffer, nameIndex, baseIndex,
                           header)
      largestReference = max(largestReference,
                             dynamicTable.toAbsolute(nameIdx))
  else:
    # Dynamic index reference
    assert(dynamicIdx)
    largestReference = max(largestReference, dynamicIdx)
    # Encode dynamicIdx, possibly with dynamicIdx above baseIndex
    encodeDynamicIndexReference(streamBuffer, dynamicIdx,
                                baseIndex)

# encode the prefix
encodeInteger(prefixBuffer, 0x00, largestReference, 8)
if baseIndex >= largestReference:
  encodeInteger(prefixBuffer, 0, baseIndex - largestReference, 7)
else:
  encodeInteger(prefixBuffer, 0x80,
                largestReference  - baseIndex, 7)

return controlBuffer, prefixBuffer + streamBuffer
```

## Appendix C. Change Log

**RFC Editor's Note:** Please remove this section prior to publication of a final version of this document.

### C.1. Since draft-ietf-quic-qpack-12

Editorial changes only

### C.2. Since draft-ietf-quic-qpack-11

Editorial changes only

### C.3. Since draft-ietf-quic-qpack-10

Editorial changes only

### C.4. Since draft-ietf-quic-qpack-09

- Decoders MUST emit Header Acknowledgements (#2939)
- Updated error code for multiple encoder or decoder streams (#2970)
- Added explicit defaults for new SETTINGS (#2974)

### C.5. Since draft-ietf-quic-qpack-08

- Endpoints are permitted to create encoder and decoder streams even if they can't use them (#2100, #2529)
- Maximum values for settings removed (#2766, #2767)

### C.6. Since draft-ietf-quic-qpack-06

- Clarify initial dynamic table capacity maximums (#2276, #2330, #2330)

### C.7. Since draft-ietf-quic-qpack-05

- Introduced the terms dynamic table capacity and maximum dynamic table capacity.
- Renamed SETTINGS_HEADER_TABLE_SIZE to SETTINGS_QPACK_MAX_TABLE_CAPACITY.

### C.8. Since draft-ietf-quic-qpack-04

- Changed calculation of Delta Base Index to avoid an illegal value (#2002, #2005)

### C.9. Since draft-ietf-quic-qpack-03

*Change HTTP settings defaults (#2038)

*Substantial editorial reorganization

### C.10. Since draft-ietf-quic-qpack-02

*Largest Reference encoded modulo MaxEntries (#1763)

*New Static Table (#1355)

*Table Size Update with Insert Count=0 is a connection error (#1762)

*Stream Cancellations are optional when SETTINGS_HEADER_TABLE_SIZE=0 (#1761)

*Implementations must handle 62 bit integers (#1760)

*Different error types for each QPACK stream, other changes to error handling (#1726)

*Preserve header field order (#1725)

*Initial table size is the maximum permitted when table is first usable (#1642)

### C.11. Since draft-ietf-quic-qpack-01

*Only header blocks that reference the dynamic table are acknowledged (#1603, #1605)

### C.12. Since draft-ietf-quic-qpack-00

*Renumbered instructions for consistency (#1471, #1472)

*Decoder is allowed to validate largest reference (#1404, #1469)

*Header block acknowledgments also acknowledge the associated largest reference (#1370, #1400)

*Added an acknowledgment for unread streams (#1371, #1400)

*Removed framing from encoder stream (#1361,#1467)

*Control streams use typed unidirectional streams rather than fixed stream IDs (#910,#1359)

### C.13. Since draft-ietf-quic-qcram-00

- Separate instruction sets for table updates and header blocks (#1235, #1142, #1141)

- Reworked indexing scheme (#1176, #1145, #1136, #1130, #1125, #1314)

- Added mechanisms that support one-pass encoding (#1138, #1320)

- Added a setting to control the number of blocked decoders (#238, #1140, #1143)

- Moved table updates and acknowledgments to dedicated streams (#1121, #1122, #1238)

## Acknowledgments

This draft draws heavily on the text of [RFC7541]. The indirect input of those authors is gratefully acknowledged, as well as ideas from:

- Ryan Hamilton

- Patrick McManus

- Kazuho Oku

- Biren Roy

- Ian Swett

- Dmitri Tikhonov

Buck's contribution was supported by Google during his employment there.

A substantial portion of Mike's contribution was supported by Microsoft during his employment there.

## Authors' Addresses

Charles 'Buck' Krasic
Netflix

Email: ckrasic@netflix.com

Mike Bishop
Akamai Technologies

Email: mbishop@evequefou.be

Alan Frindell (editor)
Facebook

Email: afrind@fb.com