Building Blocks for HTTP APIs
Internet-Draft
Intended status: Standards Track
Expires: 8 January 2026

A. Wright
7 July 2025

Byte Range PATCH
draft-ietf-httpapi-patch-byterange-03

Abstract

This document specifies a media type for PATCH payloads that overwrites a specific byte range, facilitating random access writes and segmented uploads of resources.

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 January 2026.

Table of Contents

1. Introduction

Filesystem interfaces typically provide mechanisms to write at a specific position in a file. While HTTP supports reading byte ranges using the Range header (Section 14 of [RFC9110]), this technique cannot generally be used with PUT, as the server may ignore the Content-Range header, potentially causing data corruption. By using the PATCH method with a media type that the server understands, writing to byte ranges with Content-Range semantics becomes possible, even when server support is uncertain.

This media type is intended for use in a wide variety of applications including idempotently writing to a stream, appending data to a file, overwriting specific byte ranges, or writing to multiple regions in a single operation (for example, appending audio to a recording in progress while updating metadata at the beginning of the file).

1.1. Notational Conventions

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.

This document uses ABNF as defined in [RFC5234] and imports grammar rules from [RFC8941] and [RFC9112].

For brevity, example HTTP requests or responses may add newlines or whitespace, or omit some headers necessary for message transfer.

The term "byte" is used in the [RFC9110] sense to mean "octet." Ranges are zero-indexed and inclusive. For example, "bytes 0-0" means the first byte of the document, and "bytes 1-2" is a range with two bytes, starting one byte into the document. Ranges of zero bytes are described by an address offset rather than a range. For example, "at byte 5" would separate the byte ranges 0-4 and 5-9.

2. Modifying a Content Range with PATCH

Sending a Content-Range field in a PUT request (A "partial PUT", [RFC9110], Section 14.5) requires server support to be processed correctly. Without such support, the server’s normal behavior would be to ignore the header, replacing the entire resource with just the part that changed, potentially causing corruption. To mitigate this, Content-Range may be used in conjunction with the PATCH method [RFC5789] and a media type whose semantics are to write at the specified byte offset. This document re-uses the "multipart/

byteranges" media type, and defines the "message/byterange" and "application/byteranges" media types, to opportunistically carry this field.

A byte range patch lists one or more _parts_. Each part specifies two essential components:

1. *Part fields*: a list of HTTP fields that specify metadata, including the range being written to, the length of the body, and information about the target document that cannot be listed in the PATCH headers, e.g. Content-Type (where it would describe the patch itself, rather than the document being updated).

2. *A part body*: the actual data to write to the specified location.

Each part MUST indicate a single contiguous range to be written to. Servers MUST reject byte range patches that don't contain a known range with a 422 or 400 error. (This would mean the client may be using a yet-undefined mechanism to specify the target range.)

The simplest form to represent a byte range patch is the "message/byterange" media type, which is similar to an HTTP message:

```
Content-Range: bytes 2-5/12

wxyz
```

This patch instructs the server to write the four bytes "wxyz" at an offset of 2 bytes. Given a document containing the digits 0-9 terminated by a CRLF, the result after applying the patch would be:

```
01wxyz6789竦坂衰
```

Although this example is a text document, part bodies are as binary data, and may overwrite individual bytes of multi-byte characters.

2.1. The Content-Range Field

The Content-Range field (as seen inside a patch document) is used to specify where in the target document the part body will be written.

The client MAY indicate the anticipated final size of the document by providing the complete-length form, for example the "12" in bytes 0-11/12. The complete-length does not affect the write by itself, however the server MAY use it for other purposes, especially for preallocating disk space, or deciding when an upload in multiple parts has finished.

If the client does not know or care about the final length of the document, it MAY use * in place of complete-length. For example, bytes 0-11/*. Most random access writes will take this form.

The unsatisfied-range form (e.g. bytes */1000) sets the size of the document, but without writing any data. It may be used to truncate a document (to invert an append operation), to allocate space for a document without specifying any of its contents, or to signal that an upload has ended when the previous part or request did not specify a complete-length. The part body MUST be empty.

If the "last-pos" is unknown because the upload is indeterminate length (the Content-Length of the request is not known from the start, or the upload might continue indefinitely), then the Content-Offset field must be used instead.

2.2. The Content-Length Field

A "Content-Length" part field, if provided, describes the length of the part body. (To describe the size of the entire target resource, see the Content-Range field.)

If provided, it MUST exactly match the length of the range specified in the Content-Range field, and servers MUST error when the Content-Length mismatches the length of the range provided in Content-Range.

2.3. The Content-Offset Field

The Content-Offset field specifies an offset to write the content at, when the end of the write is not known. It is used instead of Content-Range when the Content-Length is not known.

Its syntax is specified as a Structured Field [RFC8941]:

Content-Offset = sf-item

The value indicates how much of the target document is to be skipped over, typically the number of bytes. It MUST be an sf-integer.

It accepts two parameters:

The "unit" parameter indicates the unit associated with the value. A missing "unit" parameter is equivalent to providing unit=bytes.

The "complete-length" parameter is equivalent to the complete-length value in Content-Range. When provided, it MUST be an sf-integer specifying the intended final length of the document. A missing "complete-length" is equivalent to providing * in Content-Range.

2.4. The Content-Type Field

A "Content-Type" part field MUST have the same effect as if provided in a PUT request uploading the entire resource (patch applied). Its use is typically limited to resource creation.

2.5. Other Fields

Other part fields in the patch document SHOULD have the same meaning as if it is a message-level header in a PUT request uploading the entire resource (patch applied).

Use of such fields SHOULD be limited to cases where the meaning in the HTTP request headers would be different, where they would describe the entire patch, rather than the part body. For example, the "Content-Type" or "Content-Location" fields.

2.6. Applying a Patch

Servers SHOULD NOT accept requests that write beyond, and not adjacent to, the end of the resource. This would create a sparse file, where some bytes are undefined. For example, writing at byte 601 of a resource where bytes 0-599 are defined; this would leave byte 600 undefined. Servers that accept sparse writes MUST NOT disclose uninitialized content, and SHOULD fill in undefined regions with zeros.

The expected length of the write can be computed from the part fields. If the actual length of the part body mismatches the expected length, this MUST be treated the same as a network interruption at the shorter length, but anticipating the longer length. Recovering from this interruption may involve rolling back the entire request, or saving as many bytes as possible. The client can then recover as it would recover from a network interruption.

2.7. Range Units

Currently, the only defined range unit is "bytes", however this may be other, yet-to-be-defined values.

In the case of "bytes", the bytes that are read are exactly the same as the bytes that are changed. However, other units may define write semantics different from a read, if symmetric behavior would not make sense. For example, if a Content-Range field adds an item in a JSON array, this write may add a leading or trailing comma, not technically part of the item itself, in order to keep the resulting document well-formed.

Even though the length in alternate units isn't changed, the byte length might. This might only be acceptable to servers storing these values in a database or memory structure, rather than on a byte-based filesystem.

3. Byterange Media Types

3.1. The multipart/byteranges Media Type

The following illustrates a request with a "multipart/byteranges" body to write two ranges in a document:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: multipart/byteranges; boundary=THIS_STRING_SEPARATES
Content-Length: 206
If-Match: "xyzzy"
If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT

--THIS_STRING_SEPARATES
Content-Range: bytes 2-6/25
Content-Type: text/plain

23456
--THIS_STRING_SEPARATES
Content-Range: bytes 17-21/25
Content-Type: text/plain

78901
--THIS_STRING_SEPARATES--
```

The syntax for multipart messages is defined in [RFC2046], Section 5.1.1. While the body cannot contain the boundary, servers MAY use the Content-Length field to skip to the boundary (potentially ignoring a boundary in the body, which would be an error by the client). Content-Range MUST NOT be used in place of Content-Length for this purpose.

The multipart/byteranges type may be used for operations where multiple regions must be updated at the same time; clients expect all the changes to be recorded as a single operation, or that if there's an interruption, all of the parts will be rolled back together. However, the exact behavior is at the discretion of the server.

3.2. The message/byterange Media Type

When making a request with a single byte range, there is no need for a multipart boundary marker. This document defines a new media type "message/byterange" with the same semantics as a single byte range in a multipart/byteranges message, but with a simplified syntax that only needs to be parsed to the first empty line.

The "message/byterange" form may be used in a PATCH request as so:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: message/byterange
Content-Length: 272
If-Match: "xyzzy"
If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT

Content-Range: bytes 100-299/600
Content-Type: text/plain

[200 bytes...]
```

This represents a request to modify a 600-byte document, starting at a 100-byte offset, overwriting 200 bytes of it.

3.2.1. Syntax

The syntax re-uses concepts from the "multipart/byteranges" media type, except it omits the multipart separator, and so only allows a single range to be specified. It is also similar to the "message/http" media type, except the first line (the status line or request line) is omitted; a message/byterange document can be fed into a message/http parser by first prepending a line like "PATCH / HTTP/1.1".

It follows the syntax of HTTP message headers and body. It MUST include the Content-Range header field. If the message length is known by the sender, it SHOULD contain the Content-Length header field. Unknown or nonapplicable header fields MUST be ignored.

The field-line and message-body productions are specified in [RFC9112].

```
byterange-document = *( field-line CRLF )
                     CRLF
                     [ message-body ]
```

This document has the same semantics as a single part in a "multipart/byteranges" document (Section 5.1.1 of [RFC2046]) or any response with a 206 (Partial Content) status code (Section 15.3.7 of [RFC9110]). A "message/byterange" document may be trivially transformed into a "multipart/byteranges" document by prepending a dash-boundary and CRLF, and appending a close-delimiter (a CRLF, dash-boundary, terminating "--", and optional CRLF).

3.3. The application/byteranges Media Type

The "application/byteranges" has the same semantics as "multipart/byteranges" but follows a binary format similar to "message/bhttp" [RFC9292], which may be more suitable for some clients and servers, as all variable length strings are tagged with their length.

3.3.1. Syntax

Parsing starts by looking for a "Known-Length Message" or an "Indeterminate-Length Message". One or the other is distinguished by reading the Framing Indicator.

The remainder of the message is parsed by reading fields, then the content, by a method depending on if the message is known-length or indeterminate-length. If there are additional parts, they begin immediately after the end of a Content.

The "Known-Length Field Section", "Known-Length Content", "Indeterminate-Length Field Section", "Indeterminate-Length Content", "Indeterminate-Length Content Chunk", "Field Line" definitions are identical to their definition in message/bhttp. They are used in one or more Known-Length Message and/or Indeterminate-Length Message productions, concatenated together.

```
Patch {
  Message (..) ...
}

Known-Length Message {
  Framing Indicator (i) = 8,
  Known-Length Field Section (..),
  Known-Length Content (..),
}

Indeterminate-Length Message  {
  Framing Indicator (i) = 10,
  Indeterminate-Length Field Section (..),
  Indeterminate-Length Content (..),
}

Known-Length Field Section {
  Length (i),
  Field Line (..) ...,
}

Known-Length Content {
  Content Length (i),
  Content (..),
}

Indeterminate-Length Field Section {
  Field Line (..) ...,
  Content Terminator (i) = 0,
}

Indeterminate-Length Content {
  Indeterminate-Length Content Chunk (..) ...,
  Content Terminator (i) = 0,
}

Indeterminate-Length Content Chunk {
  Chunk Length (i) = 1..,
  Chunk (..),
}

Field Line {
  Name Length (i) = 1..,
  Name (..),
  Value Length (i),
  Value (..),
}
```

4. Preserving Incomplete Uploads with "Prefer: transaction"

The stateless design of HTTP generally implies that a request is atomic (otherwise parties would need to keep track of the state of a request while it's in progress). Clients need not need be concerned with the side-effects of error halfway through an upload.

However, some clients may desire partial state changes, particularly when remaking the upload is more expensive than recovering from an interruption. In these cases, clients will prefer the incomplete upload to be preserved as much as possible, so they may resume from where the incomplete request was terminated.

The client's preference for atomic or upload-preserving behavior may be signaled by a Prefer header:

```
Prefer: transaction=atomic
Prefer: transaction=persist
```

The transaction=atomic preference indicates that the request SHOULD commit only when a successful response is returned, and not any time before the end of the upload.

The transaction=persist preference indicates that uploaded data SHOULD be continuously committed, so that if the upload is interrupted, it is possible to resume the upload from where it left off.

This preference is generally applicable to any HTTP request (and not merely for PATCH or byte range patches). Servers SHOULD indicate when this preference was honored, using a "Preference-Applied" response header. For example:

```
Preference-Applied: transaction=persist
```

Servers might consider signaling this in a 103 (Early Hints) response [RFC8297], since once the final response is written, this may no longer be useful information.

5. Segmented Document Creation with PATCH

As an alternative to using PUT to create a new resource, the contents of a resource may be uploaded in segments, written across several PATCH requests.

A user-agent may also use PATCH to recover from an interrupted PUT request, if it was expected to create a new resource. The server will store the data sent to it by the user agent, but will not finalize the upload until the complete length of the document is known and received.

1. The client makes a PUT or PATCH request to a URL, a portion of which is randomized, to be unpredictable to other clients. This first request creates the resource, and should include Prefer: transaction=persist (to continuously persist the upload) and If-None-Match: * (to verify the target does not exist). If a PUT request, the server reads the Content-Length header and stores the intended complete length of the document. If a PATCH request, the "Content-Range" field in the "message/byterange" patch is read for the complete length. The complete length may also be undefined, and defined in a later request.

2. If any request is interrupted, the client may make a HEAD request to determine how much, if any, of the previous response was stored. A Content-Length response header indicates what the size of an equivalent GET request would have been. Since this will be content uploaded in the interrupted attempt, this is the offset where the next request can resume uploading from.

3. If the client sees from the HEAD response that additional data remains to be uploaded, it may make a PATCH request to resume uploading. Even if no data was uploaded or the resource was not created, the client should attempt creating the resource with PATCH to mitigate the possibility of another interrupted connection with a server that does not save incomplete transfers. However if in response to PATCH, the server reports 405 (Method Not Allowed), 415 (Unsupported Media Type), or 501 (Not Implemented), then the client must resort to retrying the PUT request.

4. The server detects the completion of the final request when the current received data matches the indicated complete length. For example, a Content-Range: 500-599/600 field is a write at the end of the resource. The server processes the upload and returns a response for it.

If the client does not know the complete length of its upload (for example, an upload of live audio), it may use the indeterminate length form "*" to append data to the document. The end of the upload is then signaled with an "unsatisfied-range" form (e.g. Content-Range: */1000), which instructs the server that the upload is complete if it has received all 1000 bytes.

For building POST endpoints that support large uploads, clients can first upload the data to a scratch file as described above, and then submit a link to the file in a POST request.

For updating an existing large file, the client can upload to a scratch file, then execute a MOVE (Section 9.9 of [RFC4918]) over the intended target.

5.1. Complete Length Example

This example shows how to upload a 600-byte document over multiple requests with PATCH, 200 bytes at a time.

The first PATCH request creates the resource:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: message/byterange
Content-Length: 281
If-None-Match: *

Content-Range: bytes 0-199/600
Content-Type: text/plain
Content-Length: 200

[200 bytes...]
```

This request allocates a 600 byte document, and uploading the first 200 bytes of it. The server responds with 200, indicating that the complete upload was stored.

Additional requests upload the remainder of the document:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: message/byterange
Content-Length: 283
If-None-Match: *

Content-Range: bytes 200-399/600
Content-Type: text/plain
Content-Length: 200

[200 bytes...]
```

This second request also returns 200 (OK).

A third request uploads the final portion of the document:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: message/byterange
Content-Length: 283
If-None-Match: *

Content-Range: bytes 200-399/600
Content-Type: text/plain
Content-Length: 200

[200 bytes...]
```

The server responds with 200 (OK). Since this completely writes out the 600-byte document, the server may also perform final processing, for example, checking that the document is well formed. The server MAY return an error code if there is a syntax or other error, or in an earlier response as soon as it it able to detect an error, however the exact behavior is left undefined.

6. Registrations

6.1. message/byterange

Type name: message

Subtype name: byterange

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See Section 7

Interoperability considerations: See Section 3.2 of this document

Published specification: This document

Applications that use this media type: HTTP applications that process filesystem-like writes to locations within a resource.

Fragment identifier considerations: N/A

Additional information: Deprecated alias names for this type: N/A
Magic number(s): N/A
File extension(s): N/A
Macintosh file type code(s): N/A

Person and email address to contact for further information: See Aut hors' Addresses section.

Intended usage: COMMON

Restrictions on usage: None.

Author: See Authors' Addresses section.

Change controller: IESG

6.2. application/byteranges

Type name: application

Subtype name: byteranges

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See Section 7

Interoperability considerations: See Section 3.3 of this document

Published specification: This document

Applications that use this media type: HTTP applications that process filesystem-like writes to locations within a resource.

Fragment identifier considerations: N/A

Additional information: Deprecated alias names for this type: N/A
Magic number(s): N/A
File extension(s): N/A
Macintosh file type code(s): N/A

Person and email address to contact for further information: See Aut hors' Addresses section.

Intended usage: COMMON

Restrictions on usage: None.

Author: See Authors' Addresses section.

Change controller: IESG

6.3. Content-Offset

Field name: Content-Offset

Status: permanent

Specification document: This document.

6.4. "transaction" preference

Preference: transaction

Value: either "atomic" or "persist"

Description: Specify if the client would prefer incomplete uploads to be saved, or committed only on success.

Reference: Section 4

7. Security Considerations

7.1. Uninitialized ranges

A byterange patch may permit writes to offsets beyond the end of the resource. This may have non-obvious behavior.

Servers supporting sparse files MUST NOT return uninitialized memory or storage contents. Uninitialized regions may be initialized prior to executing the write, or this may be left to the filesystem if it can guarantee that unallocated space will be read as a constant value.

7.2. Initializing large documents

A byte range patch typically only requires server resources proportional to the patch size. One exception is pre-initializing space when growing a file. This occurs when a complete-length is specified in the Content-Range field, which hints at the final upload size, or when writing to an offset far after the end of the file, and the server initializes the space in between. This initialization could be a very expensive operation compared to the actual size of the patch.

In general, servers SHOULD treat an initialization towards resource limits, and issue a 400 (Client Error) as appropriate. Note that 413 (Payload Too Large) would be misleading in this situation, as it

would indicate the patch itself is too large, and the client should break up the patches into smaller chunks, rather than the intended signal that the final upload size isd too large.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/rfc/rfc5234>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8941] Nottingham, M. and P. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/rfc/rfc8941>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[RFC9112] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", STD 99, RFC 9112, DOI 10.17487/RFC9112, June 2022, <https://www.rfc-editor.org/rfc/rfc9112>.

8.2. Informative References

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/rfc/rfc2046>.

[RFC4918] Dusseault, L., Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, DOI 10.17487/RFC4918, June 2007, <https://www.rfc-editor.org/rfc/rfc4918>.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/RFC5789, March 2010, <https://www.rfc-editor.org/rfc/rfc5789>.

[RFC8297] Oku, K., "An HTTP Status Code for Indicating Hints", RFC 8297, DOI 10.17487/RFC8297, December 2017, <https://www.rfc-editor.org/rfc/rfc8297>.

[RFC9292] Thomson, M. and C. A. Wood, "Binary Representation of HTTP Messages", RFC 9292, DOI 10.17487/RFC9292, August 2022, <https://www.rfc-editor.org/rfc/rfc9292>.

Appendix A. Discussion

// This section to be removed before final publication.

A.1. Sparse and Noncontiguous Resources

This pattern can enable multiple, parallel uploads to a document at the same time. For example, uploading a large log file from multiple devices. However, this document does not define any ways for clients to track the unwritten regions in sparse documents, and the existing conditional request headers are designed to cause conflicts. Parallel uploads may require a byte-level locking scheme or conflict-free operators. This may be addressed in a later document.

A.2. Recovering from interrupted PUT

Servers do not necessarily save the results of an incomplete upload; since most clients prefer atomic writes, many servers will discard an incomplete upload. A mechanism to indicate a preference for atomic vs. non-atomic writes may be defined at a later time.

Byte range PATCH cannot by itself provide recovery from an interrupted PUT to an existing document, as it is not generally possible to distinguish what was received by the server from what already existed.

One technique would be to use a 1xx interim response to indicate a location where the partial upload is being stored. If PUT request is interrupted, the client can make PATCH requests to this temporary, non-atomic location to complete the upload. When the last part is uploaded, the original interrupted PUT request will finish.

Another technique would be to send a 1xx interim response to indicate how much of the upload has been committed. When a client receives this, it can skip that many bytes from the next attempt in a PATCH request.

A.3. Splicing and Binary Diff

Operations more complicated than standard filesystem operations are out of scope for this media type. A feature of byte range patch is an upper limit on the complexity of applying the patch. In contrast, prepending, splicing, replace, or other complicated file operations could potentially require the entire file on disk be rewritten.

Consider registering a media type for VCDIFF in this document, under the topic of "Media type registrations for byte-level patching".

A.4. Ending Indeterminate Length Uploads

When uploading a patch with indeterminate-length form, Since the server doesn’t know the complete-length, it might not be able to assume that the end of the request is also the end of the whole document, even if the request ended cleanly (the client may have reasons to split apart the upload). This needs to be signaled by a subsequent request with an unsatisfied-range specifying the final length of the document.

A.5. Reading Content-Range and Content-Offset in the headers

Some parties may prefer that the message body need not be parsed at all, but instead leverage existing HTTP mechanisms to carry the part field data. Similar to how a 206 status code indicates that the response body is partial, a reserved value of Content-Type (in a PATCH request) could signal the server to read the Content-Range from the request headers, and the whole request body is the part body.

Author’s Address

Austin Wright
Email: aaa@bzfx.net