Stream: Internet Engineering Task Force (IETF)
RFC: 8792
Category: Informational
Published: June 2020
ISSN: 2070-1721
Authors:
K. Watsen, *Watsen Networks*
E. Auerswald, *Individual Contributor*
A. Farrel, *Old Dog Consulting*
Q. Wu, *Huawei Technologies*

RFC 8792
Handling Long Lines in Content of Internet-Drafts and RFCs

Abstract

This document defines two strategies for handling long lines in width-bounded text content. One strategy, called the "single backslash" strategy, is based on the historical use of a single backslash ('\') character to indicate where line-folding has occurred, with the continuation occurring with the first character that is not a space character (' ') on the next line. The second strategy, called the "double backslash" strategy, extends the first strategy by adding a second backslash character to identify where the continuation begins and is thereby able to handle cases not supported by the first strategy. Both strategies use a self-describing header enabling automated reconstitution of the original content.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

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). Not all documents approved by the IESG are candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8792.

Copyright Notice

Table of Contents

1. Introduction

[RFC7994] sets out the requirements for plain-text RFCs and states that each line of an RFC (and hence of an Internet-Draft) must be limited to 72 characters followed by the character sequence that denotes an end-of-line (EOL).

Internet-Drafts and RFCs often include example text or code fragments. Many times, the example text or code exceeds the 72-character line-length limit. The 'xml2rfc' utility [xml2rfc], at the time of this document's publication, does not attempt to wrap the content of such inclusions, simply issuing a warning whenever lines exceed 69 characters. Historically, there has been no convention recommended by the RFC Editor in place for how to handle long lines in such inclusions, other than advising authors to clearly indicate what manipulation has occurred.

This document defines two strategies for handling long lines in width-bounded text content. One strategy, called the "single backslash" strategy, is based on the historical use of a single backslash ('\') character to indicate where line-folding has occurred, with the continuation occurring with the first character that is not a space character (' ') on the next line. The second strategy, called the "double backslash" strategy, extends the first strategy by adding a second backslash character to identify where the continuation begins and is thereby able to handle cases not supported by the first strategy. Both strategies use a self-describing header enabling automated reconstitution of the original content.

The strategies defined in this document work on any text content but are primarily intended for a structured sequence of lines, such as would be referenced by the <sourcecode> element defined in Section 2.48 of [RFC7991], rather than for two-dimensional imagery, such as would be referenced by the <artwork> element defined in Section 2.5 of [RFC7991].

Note that text files are represented as lines having their first character in column 1, and a line length of N where the last character is in the Nth column and is immediately followed by an end-of-line character sequence.

2. Applicability Statement

The formats and algorithms defined in this document may be used in any context, whether for IETF documents or in other situations where structured folding is desired.

Within the IETF, this work primarily targets the xml2rfc v3 <sourcecode> element (Section 2.48 of [RFC7991]) and the xml2rfc v2 <artwork> element (Section 2.5 of [RFC7749]), which, for lack of a better option, is used in xml2rfc v2 for both source code and artwork. This work may also be used for the xml2rfc v3 <artwork> element (Section 2.5 of [RFC7991]), but as described in Section 5.1, it is generally not recommended.

3. Requirements Language

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

4. Goals

4.1. Automated Folding of Long Lines in Text Content

Automated folding of long lines is needed in order to support documents that are dynamically compiled to include content with potentially unconstrained line lengths. For instance, the build process may wish to include content from other local files or content that is dynamically generated by some external process. Both of these cases are discussed next.

Many documents need to include the content from local files (e.g., XML, JSON, ABNF, ASN.1). Prior to including a file's content, the build process **SHOULD** first validate these source files using format-specific validators. In order for such tooling to be able to process the files, the files must be in their original/natural state, which may entail them having some long lines. Thus, these source files need to be folded before inclusion into the XML document, in order to satisfy 'xml2rfc' line-length limits.

Similarly, documents sometimes contain dynamically generated output, typically from an external process operating on the same source files discussed in the previous paragraph. For instance, such processes may translate the input format to another format, or they may render a report on, or a view of, the input file. In some cases, the dynamically generated output may contain lines exceeding the 'xml2rfc' line-length limits.

In both cases, folding is required and **SHOULD** be automated to reduce effort and errors resulting from manual processing.

4.2. Automated Reconstitution of the Original Text Content

Automated reconstitution of the exact original text content is needed to support validation of text-based content extracted from documents.

For instance, YANG modules [RFC7950] are already extracted from Internet-Drafts and validated as part of the submission process. Additionally, the desire to validate instance examples (i.e., XML/JSON documents) contained within Internet-Drafts has been discussed [yang-doctors-thread].

5. Limitations

5.1. Not Recommended for Graphical Artwork

While the solution presented in this document works on any kind of text-based content, it is most useful on content that represents source code (XML, JSON, etc.) or, more generally, on content that has not been laid out in two dimensions (e.g., diagrams).

Fundamentally, the issue is whether the text content remains readable once folded. Text content that is unpredictable is especially susceptible to looking bad when folded; falling into this category are most Unified Modeling Language (UML) diagrams, YANG tree diagrams, and ASCII art in general.

It is **NOT RECOMMENDED** to use the solution presented in this document on graphical artwork.

5.2. Doesn't Work as Well as Format-Specific Options

The solution presented in this document works generically for all text-based content, as it only views content as plain text. However, various formats sometimes have built-in mechanisms that are better suited to prevent long lines.

For instance, both the 'pyang' and 'yanglint' utilities [pyang] [yanglint] have the command-line option "tree-line-length", which can be used to indicate a desired maximum line length when generating YANG tree diagrams [RFC8340].

In another example, some source formats (e.g., YANG [RFC7950]) allow any quoted string to be broken up into substrings separated by a concatenation character (e.g., '+'), any of which can be on a different line.

It is **RECOMMENDED** that authors do as much as possible within the selected format to avoid long lines.

6. Two Folding Strategies

This document defines two nearly identical strategies for folding text-based content.

The Single Backslash Strategy ('\'):
: Uses a backslash ('\') character at the end of the line where folding occurs, and assumes that the continuation begins at the first character that is not a space character (' ') on the following line.

The Double Backslash Strategy ('\\'):
: Uses a backslash ('\') character at the end of the line where folding occurs, and assumes that the continuation begins after a second backslash ('\') character on the following line.

6.1. Comparison

The first strategy produces output that is more readable. However, (1) it is significantly more likely to encounter unfoldable input (e.g., a long line containing only space characters), and (2) for long lines that can be folded, automation implementations may encounter scenarios that, without special care, will produce errors.

The second strategy produces output that is less readable, but it is unlikely to encounter unfoldable input, there are no long lines that cannot be folded, and no special care is required when folding a long line.

6.2. Recommendation

It is **RECOMMENDED** that implementations first attempt to fold content using the single backslash strategy and, only in the unlikely event that it cannot fold the input or the folding logic is unable to cope with a contingency occurring on the desired folding column, then fall back to the double backslash strategy.

7. The Single Backslash Strategy ('\')

7.1. Folded Structure

Text content that has been folded as specified by this strategy **MUST** adhere to the following structure.

7.1.1. Header

The header is two lines long.

The first line is the following 36-character string; this string **MAY** be surrounded by any number of printable characters. This first line cannot itself be folded.

```
NOTE: '\' line wrapping per RFC 8792
```

The second line is an empty line, containing only the end-of-line character sequence. This line provides visual separation for readability.

7.1.2. Body

The character encoding is the same as the encoding described in Section 2 of [RFC7994], except that, per [RFC7991], tab characters are prohibited.

Lines that have a backslash ('\') occurring as the last character in a line are considered "folded".

Exceptionally long lines MAY be folded multiple times.

7.2. Algorithm

This section describes a process for folding and unfolding long lines when they are encountered in text content.

The steps are complete, but implementations **MAY** achieve the same result in other ways.

When a larger document contains multiple instances of text content that may need to be folded or unfolded, another process must insert/extract the individual text content instances to/from the larger document prior to utilizing the algorithms described in this section. For example, the 'xiax' utility [xiax] does this.

7.2.1. Folding

Determine the desired maximum line length from input to the line-wrapping process, such as from a command-line parameter. If no value is explicitly specified, the value "69" **SHOULD** be used.

Ensure that the desired maximum line length is not less than the minimum header, which is 36 characters. If the desired maximum line length is less than this minimum, exit (this text-based content cannot be folded).

Scan the text content for horizontal tab characters. If any horizontal tab characters appear, either resolve them to space characters or exit, forcing the input provider to convert them to space characters themselves first.

Scan the text content to ensure that at least one line exceeds the desired maximum. If no line exceeds the desired maximum, exit (this text content does not need to be folded).

Scan the text content to ensure that no existing lines already end with a backslash ('\') character, as this could lead to an ambiguous result. If such a line is found, and its width is less than the desired maximum, then it **SHOULD** be flagged for "forced" folding (folding even though unnecessary). If the folding implementation doesn't support forced foldings, it **MUST** exit.

If this text content needs to, and can, be folded, insert the header described in Section 7.1.1, ensuring that any additional printable characters surrounding the header do not result in a line exceeding the desired maximum.

For each line in the text content, from top to bottom, if the line exceeds the desired maximum or requires a forced folding, then fold the line by performing the following steps:

1. Determine where the fold will occur. This location **MUST** be before or at the desired maximum column and **MUST NOT** be chosen such that the character immediately after the fold is a space (' ') character. For forced foldings, the location is between the '\' and the end-of-line sequence. If no such location can be found, then exit (this text content cannot be folded).
2. At the location where the fold is to occur, insert a backslash ('\') character followed by the end-of-line character sequence.
3. On the following line, insert any number of space (' ') characters, provided that the resulting line does not exceed the desired maximum.

The result of the previous operation is that the next line starts with an arbitrary number of space (' ') characters, followed by the character that was previously occupying the position where the fold occurred.

Continue in this manner until reaching the end of the text content. Note that this algorithm naturally addresses the case where the remainder of a folded line is still longer than the desired maximum and, hence, needs to be folded again, ad infinitum.

The process described in this section is illustrated by the "fold_it_1()" function in Appendix A.

7.2.2. Unfolding

Scan the beginning of the text content for the header described in Section 7.1.1. If the header is not present, exit (this text content does not need to be unfolded).

Remove the two-line header from the text content.

For each line in the text content, from top to bottom, if the line has a backslash ('\') character immediately followed by the end-of-line character sequence, then the line can be unfolded. Remove the backslash ('\') character, the end-of-line character sequence, and any leading space (' ') characters, which will bring up the next line. Then continue to scan each line in the text content starting with the current line (in case it was multiply folded).

Continue in this manner until reaching the end of the text content.

The process described in this section is illustrated by the "unfold_it_1()" function in Appendix A.

8. The Double Backslash Strategy ('\\')

8.1. Folded Structure

Text content that has been folded as specified by this strategy **MUST** adhere to the following structure.

8.1.1. Header

The header is two lines long.

The first line is the following 37-character string; this string **MAY** be surrounded by any number of printable characters. This first line cannot itself be folded.

```
NOTE: '\\' line wrapping per RFC 8792
```

The second line is an empty line, containing only the end-of-line character sequence. This line provides visual separation for readability.

8.1.2. Body

The character encoding is the same as the encoding described in Section 2 of [RFC7994], except that, per [RFC7991], tab characters are prohibited.

Lines that have a backslash ('\') occurring as the last character in a line immediately followed by the end-of-line character sequence, when the subsequent line starts with a backslash ('\') as the first character that is not a space character (' '), are considered "folded".

Exceptionally long lines MAY be folded multiple times.

8.2. Algorithm

This section describes a process for folding and unfolding long lines when they are encountered in text content.

The steps are complete, but implementations **MAY** achieve the same result in other ways.

When a larger document contains multiple instances of text content that may need to be folded or unfolded, another process must insert/extract the individual text content instances to/from the larger document prior to utilizing the algorithms described in this section. For example, the 'xiax' utility [xiax] does this.

8.2.1. Folding

Determine the desired maximum line length from input to the line-wrapping process, such as from a command-line parameter. If no value is explicitly specified, the value "69" **SHOULD** be used.

Ensure that the desired maximum line length is not less than the minimum header, which is 37 characters. If the desired maximum line length is less than this minimum, exit (this text-based content cannot be folded).

Scan the text content for horizontal tab characters. If any horizontal tab characters appear, either resolve them to space characters or exit, forcing the input provider to convert them to space characters themselves first.

Scan the text content to see if any line exceeds the desired maximum. If no line exceeds the desired maximum, exit (this text content does not need to be folded).

Scan the text content to ensure that no existing lines already end with a backslash ('\') character while the subsequent line starts with a backslash ('\') character as the first character that is not a space character (' '), as this could lead to an ambiguous result. If such a line is found and its width is less than the desired maximum, then it **SHOULD** be flagged for forced folding (folding even though unnecessary). If the folding implementation doesn't support forced foldings, it **MUST** exit.

If this text content needs to, and can, be folded, insert the header described in Section 8.1.1, ensuring that any additional printable characters surrounding the header do not result in a line exceeding the desired maximum.

For each line in the text content, from top to bottom, if the line exceeds the desired maximum or requires a forced folding, then fold the line by performing the following steps:

1. Determine where the fold will occur. This location **MUST** be before or at the desired maximum column. For forced foldings, the location is between the '\' and the end-of-line sequence on the first line.
2. At the location where the fold is to occur, insert a first backslash ('\') character followed by the end-of-line character sequence.

3. On the following line, insert any number of space (' ') characters, provided that the resulting line does not exceed the desired maximum, followed by a second backslash ('\') character.

The result of the previous operation is that the next line starts with an arbitrary number of space (' ') characters, followed by a backslash ('\') character, immediately followed by the character that was previously occupying the position where the fold occurred.

Continue in this manner until reaching the end of the text content. Note that this algorithm naturally addresses the case where the remainder of a folded line is still longer than the desired maximum and, hence, needs to be folded again, ad infinitum.

The process described in this section is illustrated by the "fold_it_2()" function in Appendix A.

8.2.2. Unfolding

Scan the beginning of the text content for the header described in Section 8.1.1. If the header is not present, exit (this text content does not need to be unfolded).

Remove the two-line header from the text content.

For each line in the text content, from top to bottom, if the line has a backslash ('\') character immediately followed by the end-of-line character sequence and if the next line has a backslash ('\') character as the first character that is not a space character (' '), then the lines can be unfolded. Remove the first backslash ('\') character, the end-of-line character sequence, any leading space (' ') characters, and the second backslash ('\') character, which will bring up the next line. Then, continue to scan each line in the text content starting with the current line (in case it was multiply folded).

Continue in this manner until reaching the end of the text content.

The process described in this section is illustrated by the "unfold_it_2()" function in Appendix A.

9. Examples

The following self-documenting examples illustrate folded text-based content.

The source text content cannot be presented here, as it would again be folded. Alas, only the results can be provided.

9.1. Example Showing Boundary Conditions

This example illustrates boundary conditions. The input contains seven lines, each line one character longer than the previous line. Numbers are used for counting purposes. The default desired maximum column value "69" is used.

9.1.1. Using '\'

```
========== NOTE: '\' line wrapping per RFC 8792 ===========

1234567890123456789012345678901234567890123456789012345678901234567890123456
1234567890123456789012345678901234567890123456789012345678901234567890123456
1234567890123456789012345678901234567890123456789012345678901234567890123456
1234567890123456789012345678901234567890123456789012345678901234567890123456
1234567890123456789012345678901234567890123456789012345678901234567890123456
90
1234567890123456789012345678901234567890123456789012345678901234567890123456
901
1234567890123456789012345678901234567890123456789012345678901234567890123456
9012
```

9.1.2. Using '\\'

```
========== NOTE: '\\' line wrapping per RFC 8792 ==========

1234567890123456789012345678901234567890123456789012345678901234567890123456
1234567890123456789012345678901234567890123456789012345678901234567890123456
1234567890123456789012345678901234567890123456789012345678901234567890123456
1234567890123456789012345678901234567890123456789012345678901234567890123456
1234567890123456789012345678901234567890123456789012345678901234567890123456
\90
1234567890123456789012345678901234567890123456789012345678901234567890123456
\901
1234567890123456789012345678901234567890123456789012345678901234567890123456
\9012
```

9.2. Example Showing Multiple Wraps of a Single Line

This example illustrates what happens when a very long line needs to be folded multiple times. The input contains one line containing 280 characters. Numbers are used for counting purposes. The default desired maximum column value "69" is used.

9.2.1. Using '\'

```
========== NOTE: '\' line wrapping per RFC 8792 ===========

1234567890123456789012345678901234567890123456789012345678901234567890123456
9012345678901234567890123456789012345678901234567890123456789012345678901234
7890123456789012345678901234567890123456789012345678901234567890123456789012
5678901234567890123456789012345678901234567890123456789012345678901234567890
34567890
```

9.2.2. Using '\\'

```
========== NOTE: '\\' line wrapping per RFC 8792 ==========

1234567890123456789012345678901234567890123456789012345678\
\9012345678901234567890123456789012345678901234567890123456789012345\
\6789012345678901234567890123456789012345678901234567890123456789012\
\3456789012345678901234567890123456789012345678901234567890123456789\
\01234567890
```

9.3. Example Showing "Smart" Folding

This example illustrates how readability can be improved via "smart" folding, whereby folding occurs at format-specific locations and format-specific indentations are used.

The text content was manually folded, since the script in Appendix A does not implement smart folding.

Note that the headers are surrounded by different printable characters than those shown in the script-generated examples.

9.3.1. Using '\'

```
[NOTE: '\' line wrapping per RFC 8792]

<yang-library
    xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"
    xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">

  <module-set>
    <name>config-modules</name>
    <module>
      <name>ietf-interfaces</name>
      <revision>2018-02-20</revision>
      <namespace>\
        urn:ietf:params:xml:ns:yang:ietf-interfaces\
      </namespace>
    </module>
    ...
  </module-set>
  ...
</yang-library>
```

Below is the equivalent of the above, but it was folded using the script in Appendix A.

```
========== NOTE: '\' line wrapping per RFC 8792 ===========

<yang-library
    xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"
    xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">

  <module-set>
    <name>config-modules</name>
    <module>
      <name>ietf-interfaces</name>
      <revision>2018-02-20</revision>
      <namespace>urn:ietf:params:xml:ns:yang:ietf-interfaces</namesp\
ace>
    </module>
    ...
  </module-set>
  ...
</yang-library>
```

9.3.2. Using '\\'

```
[NOTE: '\\' line wrapping per RFC 8792]

<yang-library
    xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"
    xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">

  <module-set>
    <name>config-modules</name>
    <module>
      <name>ietf-interfaces</name>
      <revision>2018-02-20</revision>
      <namespace>\
        \urn:ietf:params:xml:ns:yang:ietf-interfaces\
      \</namespace>
    </module>
    ...
  </module-set>
  ...
</yang-library>
```

Below is the equivalent of the above, but it was folded using the script in Appendix A.

```
========== NOTE: '\\' line wrapping per RFC 8792 ==========

<yang-library
    xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"
    xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">

  <module-set>
    <name>config-modules</name>
    <module>
      <name>ietf-interfaces</name>
      <revision>2018-02-20</revision>
      <namespace>urn:ietf:params:xml:ns:yang:ietf-interfaces</namesp\
\ace>
    </module>
    ...
  </module-set>
  ...
</yang-library>
```

9.4. Example Showing "Forced" Folding

This example illustrates how invalid sequences in lines that do not have to be folded can be handled via forced folding, whereby the folding occurs even though unnecessary.

```
The following line exceeds a 68-char max and, thus, demands folding:
1234567890123456789012345678901234567890123456789012345678901234567890123456789

This line ends with a backslash \

This line ends with a backslash \
\ This line begins with a backslash

The following is an indented 3x3 block of backslashes:
   \\\
   \\\
   \\\
```

The samples below were manually folded, since the script in the appendix does not implement forced folding.

Note that the headers are prefixed by a pound ('#') character, rather than surrounded by 'equals' ('=') characters as shown in the script-generated examples.

9.4.1. Using '\'

```
# NOTE: '\' line wrapping per RFC 8792

The following line exceeds a 68-char max and, thus, demands folding:
1234567890123456789012345678901234567890123456789012345678901234567\
89

This line ends with a backslash \\


This line ends with a backslash \\

\ This line begins with a backslash

The following is an indented 3x3 block of backslashes:
   \\\\

   \\\\

   \\\
```

9.4.2. Using '\\'

```
# NOTE: '\\' line wrapping per RFC 8792

The following line exceeds a 68-char max and, thus, demands folding:
1234567890123456789012345678901234567890123456789012345678901234567\
\89

This line ends with a backslash \

This line ends with a backslash \\
\
\ This line begins with a backslash

The following is an indented 3x3 block of backslashes:
   \\\\
   \
   \\\\
   \
   \\\
```

10. Security Considerations

This document has no security considerations.

11. IANA Considerations

This document has no IANA actions.

12. References

12.1. Normative References

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

[RFC7991] Hoffman, P., "The "xml2rfc" Version 3 Vocabulary", RFC 7991, DOI 10.17487/RFC7991, December 2016, <https://www.rfc-editor.org/info/rfc7991>.

[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>.

12.2. Informative References

[bash] "GNU Bash Manual", <https://www.gnu.org/software/bash/manual>.

[pyang] "pyang", <https://pypi.org/project/pyang/>.

[RFC7749] Reschke, J., "The "xml2rfc" Version 2 Vocabulary", RFC 7749, DOI 10.17487/RFC7749, February 2016, <https://www.rfc-editor.org/info/rfc7749>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC7994] Flanagan, H., "Requirements for Plain-Text RFCs", RFC 7994, DOI 10.17487/RFC7994, December 2016, <https://www.rfc-editor.org/info/rfc7994>.

[RFC8340] Bjorklund, M. and L. Berger, Ed., "YANG Tree Diagrams", BCP 215, RFC 8340, DOI 10.17487/RFC8340, March 2018, <https://www.rfc-editor.org/info/rfc8340>.

[xiax] "The 'xiax' Python Package", <https://pypi.org/project/xiax/>.

[xml2rfc] "xml2rfc", <https://pypi.org/project/xml2rfc/>.

[yang-doctors-thread] Watsen, K., "[yang-doctors] automating yang doctor reviews", message to the yang-doctors mailing list, 18 April 2018, <https://mailarchive.ietf.org/arch/msg/yang-doctors/DCfBqgfZPAD7afzeDFlQ1Xm2X3g>.

[yanglint] "yanglint", commit 1b7d73d, February 2020, <https://github.com/CESNET/libyang#yanglint>.

Appendix A. Bash Shell Script: rfcfold

This non-normative appendix includes a Bash shell script [bash] that can both fold and unfold text content using both the single and double backslash strategies described in Sections 7 and 8, respectively. This shell script, called 'rfcfold', is maintained at <https://github.com/ietf-tools/rfcfold>.

This script is intended to be applied to a single text content instance. If it is desired to fold or unfold text content instances within a larger document (e.g., an Internet-Draft or RFC), then another tool must be used to extract the content from the larger document before utilizing this script.

For readability purposes, this script forces the minimum supported line length to be eight characters longer than the raw header text defined in Sections 7.1.1 and 8.1.1 so as to ensure that the header can be wrapped by a space (' ') character and three 'equals' ('=') characters on each side of the raw header text.

When a tab character is detected in the input file, this script exits with the following error message:

> Error: infile contains a tab character, which is not allowed.

This script tests for the availability of GNU awk (gawk), in order to test for ASCII-based control characters and non-ASCII characters in the input file (see below). Note that testing revealed flaws in the default version of 'awk' on some platforms. As this script uses 'gawk' only to issue warning messages, if 'gawk' is not found, this script issues the following debug message:

> Debug: no GNU awk; skipping checks for special characters.

When 'gawk' is available (see above) and ASCII-based control characters are detected in the input file, this script issues the following warning message:

> Warning: infile contains ASCII control characters (unsupported).

When 'gawk' is available (see above) and non-ASCII characters are detected in the input file, this script issues the following warning message:

> Warning: infile contains non-ASCII characters (unsupported).

This script does not implement the whitespace-avoidance logic described in Section 7.2.1. In such a case, the script will exit with the following error message:

> Error: infile has a space character occurring on the folding column. This file cannot be folded using the '\' strategy.

While this script can unfold input that contains forced foldings, it is unable to fold files that would require forced foldings. Forced folding is described in Sections 7.2.1 and 8.2.1. When being asked to fold a file that would require forced folding, the script will instead exit with one of the following error messages:

For '\':

> Error: infile has a line ending with a '\' character. This file cannot be folded using the '\' strategy without there being false positives produced in the unfolding (i.e., this script does not force-fold such lines, as described in RFC 8792).

For '\\':

> Error: infile has a line ending with a '\' character followed by a '\' character as the first non-space character on the next line. This script cannot fold this file using the '\\' strategy without there being false positives produced in the unfolding (i.e., this script does not force-fold such lines, as described in RFC 8792).

Shell-level end-of-line backslash ('\') characters have been purposely added to the script so as to ensure that the script is itself not folded in this document, thus simplifying the ability to copy/paste the script for local use. As should be evident by the lack of the mandatory header described in Section 7.1.1, these backslashes do not designate a folded line (e.g., as described in Section 7).

```
<CODE BEGINS> file "rfcfold"

#!/bin/bash --posix

# This script may need some adjustments to work on a given system.
# For instance, the utility 'gsed' may need to be installed.
# Also, please be advised that 'bash' (not 'sh') must be used.

# Copyright (c) 2020 IETF Trust, Kent Watsen, and Erik Auerswald.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
#   * Redistributions of source code must retain the above copyright
#     notice, this list of conditions and the following disclaimer.
#
#   * Redistributions in binary form must reproduce the above
#     copyright notice, this list of conditions and the following
#     disclaimer in the documentation and/or other materials
#     provided with the distribution.
#
#   * Neither the name of Internet Society, IETF or IETF Trust, nor
#     the names of specific contributors, may be used to endorse or
#     promote products derived from this software without specific
#     prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
# (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
# SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
# HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
# STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
# ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

print_usage() {
  printf "\n"
  printf "Folds or unfolds the input text file according to"
  printf " RFC 8792.\n"
  printf "\n"
  printf "Usage: rfcfold [-h] [-d] [-q] [-s <strategy>] [-c <col>]"
  printf " [-r] -i <infile> -o <outfile>\n"
  printf "\n"
  printf "  -s: strategy to use, '1' or '2' (default: try 1,"
  printf " else 2)\n"
  printf "  -c: column to fold on (default: 69)\n"
  printf "  -r: reverses the operation\n"
  printf "  -i: the input filename\n"
  printf "  -o: the output filename\n"
  printf "  -d: show debug messages (unless -q is given)\n"
  printf "  -q: quiet (suppress error and debug messages)\n"
```

```
  printf "  -h: show this message\n"
  printf "\n"
  printf "Exit status code: 1 on error, 0 on success, 255 on no-op."
  printf "\n\n"
}

# global vars, do not edit
strategy=0 # auto
debug=0
quiet=0
reversed=0
infile=""
outfile=""
maxcol=69  # default, may be overridden by param
col_gvn=0  # maxcol overridden?
hdr_txt_1="NOTE: '\\' line wrapping per RFC 8792"
hdr_txt_2="NOTE: '\\\\' line wrapping per RFC 8792"
equal_chars="============================================================"
space_chars="                                                            "
temp_dir=""
prog_name='rfcfold'

# functions for diagnostic messages
prog_msg() {
  if [[ "$quiet" -eq 0 ]]; then
    format_string="${prog_name}: $1: %s\n"
    shift
    printf -- "$format_string" "$*" >&2
  fi
}

err() {
  prog_msg 'Error' "$@"
}

warn() {
  prog_msg 'Warning' "$@"
}

dbg() {
  if [[ "$debug" -eq 1 ]]; then
    prog_msg 'Debug' "$@"
  fi
}

# determine name of [g]sed binary
type gsed > /dev/null 2>&1 && SED=gsed || SED=sed

# warn if a non-GNU sed utility is used
"$SED" --version < /dev/null 2> /dev/null | grep -q GNU || \
warn 'not using GNU `sed` (likely cause if an error occurs).'

cleanup() {
  rm -rf "$temp_dir"
}
trap 'cleanup' EXIT

fold_it_1() {
```

```
  # ensure input file doesn't contain the fold-sequence already
  if [[ -n "$("$SED" -n '/\\$/p' "$infile")" ]]; then
    err "infile '$infile' has a line ending with a '\\' character."\
        "This script cannot fold this file using the '\\' strategy"\
        "without there being false positives produced in the"\
        "unfolding."
    return 1
  fi

  # where to fold
  foldcol=$(expr "$maxcol" - 1) # for the inserted '\' char

  # ensure input file doesn't contain whitespace on the fold column
  grep -q "^\(.\{$foldcol\}\)\{1,\} " "$infile"
  if [[ $? -eq 0 ]]; then
    err "infile '$infile' has a space character occurring on the"\
        "folding column.  This file cannot be folded using the"\
        "'\\' strategy."
    return 1
  fi

  # center header text
  length=$(expr ${#hdr_txt_1} + 2)
  left_sp=$(expr \( "$maxcol" - "$length" \) / 2)
  right_sp=$(expr "$maxcol" - "$length" - "$left_sp")
  header=$(printf "%.*s %s %.*s" "$left_sp" "$equal_chars"\
                   "$hdr_txt_1" "$right_sp" "$equal_chars")

  # generate outfile
  echo "$header" > "$outfile"
  echo "" >> "$outfile"
  "$SED" 's/\(.\{'"$foldcol"'\}\)\(..\)/\1\\\n\2/;t M;b;:M;P;D;'\
    < "$infile" >> "$outfile" 2> /dev/null
  if [[ $? -ne 0 ]]; then
    return 1
  fi
  return 0
}

fold_it_2() {
  # where to fold
  foldcol=$(expr "$maxcol" - 1) # for the inserted '\' char

  # ensure input file doesn't contain the fold-sequence already
  if [[ -n "$("$SED" -n '/\\$/{N;s/\\\n[ ]*\\/&/p;D}' "$infile")" ]]
  then
    err "infile '$infile' has a line ending with a '\\' character"\
        "followed by a '\\' character as the first non-space"\
        "character on the next line.  This script cannot fold"\
        "this file using the '\\\\' strategy without there being"\
        "false positives produced in the unfolding."
    return 1
  fi

  # center header text
  length=$(expr ${#hdr_txt_2} + 2)
  left_sp=$(expr \( "$maxcol" - "$length" \) / 2)
  right_sp=$(expr "$maxcol" - "$length" - "$left_sp")
```

```
  header=$(printf "%.*s %s %.*s" "$left_sp" "$equal_chars"\
                   "$hdr_txt_2" "$right_sp" "$equal_chars")

  # generate outfile
  echo "$header" > "$outfile"
  echo "" >> "$outfile"
  "$SED" 's/\(.\{'"$foldcol"'\}\)\(..\)/\1\\\n\\\2/;t M;b;:M;P;D;'\
    < "$infile" >> "$outfile" 2> /dev/null
  if [[ $? -ne 0 ]]; then
    return 1
  fi
  return 0
}

fold_it() {
  # ensure input file doesn't contain a tab
  grep -q $'\t' "$infile"
  if [[ $? -eq 0 ]]; then
    err "infile '$infile' contains a tab character, which is not"\
        "allowed."
    return 1
  fi

  # folding of input containing ASCII control or non-ASCII characters
  # may result in a wrong folding column and is not supported
  if type gawk > /dev/null 2>&1; then
    env LC_ALL=C gawk '/[\000-\014\016-\037\177]/{exit 1}' "$infile"\
    || warn "infile '$infile' contains ASCII control characters"\
            "(unsupported)."
    env LC_ALL=C gawk '/[^\000-\177]/{exit 1}' "$infile"\
    || warn "infile '$infile' contains non-ASCII characters"\
            "(unsupported)."
  else
    dbg "no GNU awk; skipping checks for special characters."
  fi

  # check if file needs folding
  testcol=$(expr "$maxcol" + 1)
  grep -q ".\{$testcol\}" "$infile"
  if [[ $? -ne 0 ]]; then
    dbg "nothing to do; copying infile to outfile."
    cp "$infile" "$outfile"
    return 255
  fi

  if [[ "$strategy" -eq 1 ]]; then
    fold_it_1
    return $?
  fi
  if [[ "$strategy" -eq 2 ]]; then
    fold_it_2
    return $?
  fi
  quiet_sav="$quiet"
  quiet=1
  fold_it_1
  result=$?
  quiet="$quiet_sav"
```

```
  if [[ "$result" -ne 0 ]]; then
    dbg "Folding strategy '1' didn't succeed; trying strategy '2'..."
    fold_it_2
    return $?
  fi
  return 0
}

unfold_it_1() {
  temp_dir=$(mktemp -d)

  # output all but the first two lines (the header) to wip file
  awk "NR>2" "$infile" > "$temp_dir/wip"

  # unfold wip file
  "$SED" '{H;$!d};x;s/^\n//;s/\\\n *//g' "$temp_dir/wip" > "$outfile"

  return 0
}

unfold_it_2() {
  temp_dir=$(mktemp -d)

  # output all but the first two lines (the header) to wip file
  awk "NR>2" "$infile" > "$temp_dir/wip"

  # unfold wip file
  "$SED" '{H;$!d};x;s/^\n//;s/\\\n *\\//g' "$temp_dir/wip"\
    > "$outfile"

  return 0
}

unfold_it() {
  # check if file needs unfolding
  line=$(head -n 1 "$infile")
  line2=$("$SED" -n '2p' "$infile")
  result=$(echo "$line" | fgrep "$hdr_txt_1")
  if [[ $? -eq 0 ]]; then
    if [[ -n "$line2" ]]; then
      err "the second line in '$infile' is not empty."
      return 1
    fi
    unfold_it_1
    return $?
  fi
  result=$(echo "$line" | fgrep "$hdr_txt_2")
  if [[ $? -eq 0 ]]; then
    if [[ -n "$line2" ]]; then
      err "the second line in '$infile' is not empty."
      return 1
    fi
    unfold_it_2
    return $?
  fi
  dbg "nothing to do; copying infile to outfile."
  cp "$infile" "$outfile"
  return 255
```

```
}

process_input() {
  while [[ "$1" != "" ]]; do
    if [[ "$1" == "-h" ]] || [[ "$1" == "--help" ]]; then
      print_usage
      exit 0
    elif [[ "$1" == "-d" ]]; then
      debug=1
    elif [[ "$1" == "-q" ]]; then
      quiet=1
    elif [[ "$1" == "-s" ]]; then
      if [[ "$#" -eq "1" ]]; then
        err "option '-s' needs an argument (use -h for help)."
        exit 1
      fi
      strategy="$2"
      shift
    elif [[ "$1" == "-c" ]]; then
      if [[ "$#" -eq "1" ]]; then
        err "option '-c' needs an argument (use -h for help)."
        exit 1
      fi
      col_gvn=1
      maxcol="$2"
      shift
    elif [[ "$1" == "-r" ]]; then
      reversed=1
    elif [[ "$1" == "-i" ]]; then
      if [[ "$#" -eq "1" ]]; then
        err "option '-i' needs an argument (use -h for help)."
        exit 1
      fi
      infile="$2"
      shift
    elif [[ "$1" == "-o" ]]; then
      if [[ "$#" -eq "1" ]]; then
        err "option '-o' needs an argument (use -h for help)."
        exit 1
      fi
      outfile="$2"
      shift
    else
      warn "ignoring unknown option '$1'."
    fi
    shift
  done

  if [[ -z "$infile" ]]; then
    err "infile parameter missing (use -h for help)."
    exit 1
  fi

  if [[ -z "$outfile" ]]; then
    err "outfile parameter missing (use -h for help)."
    exit 1
  fi
```

```
  if [[ ! -f "$infile" ]]; then
    err "specified file '$infile' does not exist."
    exit 1
  fi

  if [[ "$col_gvn" -eq 1 ]] && [[ "$reversed" -eq 1 ]]; then
    warn "'-c' option ignored when unfolding (option '-r')."
  fi

  if [[ "$strategy" -eq 0 ]] || [[ "$strategy" -eq 2 ]]; then
    min_supported=$(expr ${#hdr_txt_2} + 8)
  else
    min_supported=$(expr ${#hdr_txt_1} + 8)
  fi
  if [[ "$maxcol" -lt "$min_supported" ]]; then
    err "the folding column cannot be less than $min_supported."
    exit 1
  fi

  # this is only because the code otherwise runs out of equal_chars
  max_supported=$(expr ${#equal_chars} + 1 + ${#hdr_txt_1} + 1\
       + ${#equal_chars})
  if [[ "$maxcol" -gt "$max_supported" ]]; then
    err "the folding column cannot be more than $max_supported."
    exit 1
  fi
}

main() {
  if [[ "$#" -eq "0" ]]; then
     print_usage
     exit 1
  fi

  process_input "$@"

  if [[ "$reversed" -eq 0 ]]; then
    fold_it
    code=$?
  else
    unfold_it
    code=$?
  fi
  exit "$code"
}

main "$@"
<CODE ENDS>
```

Acknowledgements

The authors thank the RFC Editor for confirming that there was previously no set convention, at the time of this document's publication, for handling long lines in source code inclusions, thus instigating this work.

The authors thank the following folks for their various contributions while producing this document (sorted by first name): Ben Kaduk, Benoit Claise, Gianmarco Bruno, Italo Busi, Joel Jaeggli, Jonathan Hansford, Lou Berger, Martin Bjorklund, and Rob Wilton.

Authors' Addresses

Kent Watsen
Watsen Networks
Email: kent+ietf@watsen.net

Erik Auerswald
Individual Contributor
Email: auerswal@unix-ag.uni-kl.de

Adrian Farrel
Old Dog Consulting
Email: adrian@olddog.co.uk

Qin Wu
Huawei Technologies
Email: bill.wu@huawei.com