[MS-OXRTFCP]:
Rich Text Format (RTF) Compression Algorithm

documentation for past or current releases as specifically noted in the document, as applicable; it is preliminary documentation for the pre-release (beta) versions. Microsoft will release final documentation in connection with the commercial release of the updated or new version of this technology. As the documentation may change between this preliminary version and the final version of this technology, there are risks in relying on preliminary documentation. To the extent that you incur additional development obligations or any other costs as a result of relying on this preliminary documentation, you do so at your own risk.

Revision Summary

Date	Revision History	Revision Class	Comments
04/04/2008	0.1		Initial Availability.
04/25/2008	0.2		Revised and updated property names and other technical content.
06/27/2008	1.0		Initial Release.
08/06/2008	1.01		Updated references to reflect date of initial release.
09/03/2008	1.02		Revised and edited technical content.
12/03/2008	1.03		Updated IP notice.
03/04/2009	1.04		Revised and edited technical content.
04/10/2009	2.0		Updated applicable product releases.
07/15/2009	3.0	Major	Revised and edited for technical content.
11/04/2009	3.1	Minor	Updated the technical content.
02/10/2010	3.2	Minor	Updated the technical content.
05/05/2010	3.3	Minor	Updated the technical content.
08/04/2010	3.4	Minor	Clarified the meaning of the technical content.
11/03/2010	3.4	No change	No changes to the meaning, language, or formatting of the technical content.
03/18/2011	3.4	No change	No changes to the meaning, language, or formatting of the technical content.
08/05/2011	4.0	Major	Significantly changed the technical content.
10/07/2011	4.0	No change	No changes to the meaning, language, or formatting of the technical content.
01/20/2012	5.0	Major	Significantly changed the technical content.
04/27/2012	6.0	Major	Significantly changed the technical content.

Table of Contents

1 Introduction

The Rich Text Format (RTF) Compression Algorithm is used to compress and decompress **RTF** data, as described in [MSFT-RTF], to or from one of the supported compression formats.

Section 2 of this specification is normative and can contain the terms MAY, SHOULD, MUST, MUST NOT, and SHOULD NOT as defined in RFC 2119. Section 1.6 is also normative but cannot contain those terms. All other sections and examples in this specification are informative.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

ASCII
Augmented Backus-Naur Form (ABNF)
big-endian
cyclic redundancy check (CRC)
little-endian

The following terms are defined in [MS-OXGLOS]:

Message object
Rich Text Format (RTF)

The following terms are specific to this document:

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

References to Microsoft Open Specifications documentation do not include a publishing year because links are to the latest version of the documents, which are updated frequently. References to other documents include a publishing year when one is available.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[MS-DTYP] Microsoft Corporation, "Windows Data Types".

[MS-OXPROPS] Microsoft Corporation, "Exchange Server Protocols Master Property List".

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

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

1.2.2 Informative References

[MSFT-RTF] Microsoft Corporation, "Rich Text Format (RTF) Specification", version 1.9.1, March 2008, http://www.microsoft.com/downloads/details.aspx?FamilyID=DD422B8D-FF06-4207-B476-6B5396A18A2B&displaylang=en

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary".

[MS-OXGLOS] Microsoft Corporation, "Exchange Server Protocols Master Glossary".

1.3 Overview

This algorithm enables an implementer to compress or decompress RTF-encoded text. During compression, the RTF-encoded text is compared to a dictionary of RTF control words. If any of the input text matches the control words in the dictionary, a dictionary reference is written to the output buffer in place of the control word to reduce the bytes sent over the wire. Any content that does not have a dictionary match is simply written to the output buffer.

Conversely, during decompression, the compressed RTF-encoded text is compared against the dictionary and dictionary references are replaced with RTF control words. This algorithm defines the manner in which the RTF-encoded text is compared to the dictionary content and how the RTF-encoded text is read from the input buffer or written to the output buffer.

1.4 Relationship to Protocols and Other Algorithms

The RTF text encoding format is described in [MSFT-RTF]. This algorithm requires no additional protocols or algorithms to accomplish the compression format described in this specification. The **PidTagRtfCompressed** property ([MS-OXPROPS] section 2.1008) relies on this algorithm.

1.5 Applicability Statement

This algorithm is specifically used with information from the **PidTagRtfCompressed** property ([MS-OXPROPS] section 2.1008) of the **Message object**. Clients that do not implement this algorithm are unable to interpret the data that is packed with this algorithm. This algorithm can be used to compress and decompress any content (not just RTF). In addition, this algorithm supports the storing of content in an uncompressed form.

1.6 Standards Assignments

None.

2 Algorithm Details

2.1 Common Algorithm Details

2.1.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this algorithm. The described organization is provided to facilitate the explanation of how the algorithm behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

The following elements are specific to this algorithm:

- **Writer:** Software that writes compressed RTF data.
- **Reader:** Software that is capable of reading compressed RTF data and decompressing it into RTF encrypted text.
- **Dictionary:** A 4096-byte circular array of RTF control words. References to the dictionary are used to compress or decompress RTF data.
- **CRC Lookup Table:** A pre-computed table used for **CRC** field generation, as specified in section 2.1.2.2.1.

2.1.2 Initialization

2.1.2.1 Dictionary

The writer MUST initialize the dictionary (starting at offset 0) with the following **ASCII** string:

```
{\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil<SP>\froman<SP>\fswiss<SP>\fmodern<SP>\fsc
ript<SP>\fdecor<SP>MS<SP>Sans<SP>SerifSymbolArialTimes<SP>New<SP>RomanCourier{\colortbl\red0\
green0\blue0<CR><LF>\par<SP>\pard\plain\f0\fs20\b\i\u\tab\tx
```

where:

<SP> designates a space (ASCII value 0x20)

<CR> designates a carriage return (ASCII value 0x0d)

<LF> designates a line feed (ASCII value 0x0a)

After the dictionary is initialized, the writer MUST set the write offset and the end offset of the dictionary to 207 (pointing to the byte that follows the pre-loaded string).

Note The dictionary will not be used when the value of the **COMPTYPE** field is set to UNCOMPRESSED, as specified in section 2.1.3.1.1.

2.1.2.2 CRC

The writer MUST initialize the value of the **CRC** field, as specified in section 2.1.3.1.1, which contains a **cyclic redundancy check (CRC)**, to zero.

2.1.2.2.1 CRC Lookup Table

The pre-computed table used for generating the value of the **CRC** field, as specified in section 2.1.3.1.1, MUST contain the following 256 **DWORDs** ([MS-DTYP]):

```
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba,
0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec,
0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940,
0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116,
0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a,
0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818,
0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c,
0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086,
0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4,
0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad,
0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe,
0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252,
0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60,
0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04,
0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a,
0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e,
0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
```

```
0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0,
0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6,
0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
```

2.1.3 Processing Rules

2.1.3.1 RTF Compression Format

Unless otherwise specified, sizes in this section are expressed in **BYTES** ([MS-DTYP]), and multiple-byte values are stored in **little-endian** format.

2.1.3.1.1 RTF Compression ABNF Grammar

This section uses **Augmented Backus-Naur Form (ABNF)**, as specified in [RFC5234], to define the format of the contents stored in the **PidTagRtfCompressed** property ([MS-OXPROPS] section 2.1008).

```
RTFCOMPRESSED=Header CONTENTS

Header=COMPSIZE RAWSIZE COMPTYPE CRC     ; The size of the Header field is
                                         ; 16 (0x0010) bytes.

COMPSIZE =DWORD            ; Writers MUST set the COMPSIZE field to
                           ; the length of the compressed data
                           ; (the CONTENTS field) in bytes
                           ; plus 12 (the count of the
                           ; remaining bytes from the header).

RAWSIZE =DWORD                           ; The size in bytes of the
                                         ; uncompressed content.
COMPTYPE=COMPRESSED / UNCOMPRESSED       ; The type of compression.
COMPRESSED =%x4C.5A.46.75                ; Value of 0x75465A4C.
UNCOMPRESSED=%x4D.45.4C.41               ; Value of 0x414C454D.

CRC =DWORD                 ; If the COMPTYPE field is set to
                           ; COMPRESSED, then the CRC field is
                           ; computed from the CONTENTS field.
                           ; If the COMPTYPE field is set to
                           ; UNCOMPRESSED, then the  RC field
                           ; MUST be set to %x00.00.00.00.

CONTENTS=RAWDATA / COMPRESSEDDATA        ; The CONTENTS field is set
                                         ; to RAWDATA if the COMPTYPE
                                         ; field is set to UNCOMPRESSED.
                                         ; The CONTENTS field is set
                                         ; to COMPRESSEDDATA if the COMPTYPE
                                         ;  field is set to COMPRESSED.

RAWDATA=*LITERAL
```

```
COMPRESSEDDATA=[*RUN] ENDRUN [PADDING]
RUN=CONTROL 8*8TOKEN
ENDRUN=CONTROL 1*8TOKEN
CONTROL= OCTET
Token=REFERENCE / LITERAL
REFERENCE=WORD                          ; Value is in big-endian format.
LITERAL=OCTET
PADDING=*OCTET
```

2.1.3.1.2 Compressed RTF

The content of the compressed RTF, as specified by the **RTFCOMPRESSED** field in section 2.1.3.1.1, consists of a header and a series of runs. The number of runs varies based on the quantity of content that is compressed and sizes of the matches in the dictionary, as specified in section 2.1.2.1, and illustrated in the following diagram.

Header	**RUN1**	**RUN2**	**RUN3**	**RUN4**	**...**	**ENDRUN**	**PADDING**

The ABNF grammar specified in section 2.1.3.1.1 contains necessary details that are supplementary to the constructs defined in this section.

2.1.3.1.3 Compressed Run

A run is composed of a control byte and eight variable-sized tokens, and are defined in section 2.1.3.1.1 as the **RUN** field and the **CONTROL** field respectively. The final run, as specified by the **ENDRUN** field in section 2.1.3.1.1, can contain fewer than eight tokens.

CONTROL	TOKEN1	TOKEN2	TOKEN3	TOKEN4	TOKEN5	TOKEN6	TOKEN7	TOKEN8
1 Byte	Varies	Varies	Varies	Varies	Varies	Varies	Varies	Varies

Tokens are either a dictionary reference, as specified in section 2.1.3.1.5, or literals, depending on the value of the corresponding bit in the control byte.

Each control byte, as specified by the **CONTROL** field in section 2.1.3.1.1, contains details about how to interpret the next eight tokens. The low bit (bitmask %x1) in the **CONTROL** field corresponds to Token1, the second bit (bitmask %x2) corresponds to Token2, and so on. In the **ENDRUN** field, the bits in the **CONTROL** field after the completion dictionary reference are undefined and MUST be ignored.

The type of token, as specified by the **TOKEN** field in section 2.1.3.1.1, and its meaning depend on the value of the corresponding bit in the **CONTROL** field, as follows:

- If the bit in the **CONTROL** field is zero, then the corresponding token is a 1-byte literal that represents the exact byte in the uncompressed content.
- If the bit in the **CONTROL** field is 1, then the corresponding token is a 1-byte dictionary reference that indicates the offset and length of a series of bytes in the dictionary that corresponds to the bytes in the uncompressed content. For more details about dictionary references, see section 2.1.3.1.5.

2.1.3.1.4 Dictionary

This algorithm uses a dictionary that behaves as a 4096-byte circular array. When advancing a read or write position within the dictionary, a reference beyond the last index of the array wraps to a reference to the first byte and then advances from there.

The dictionary conceptually has a write offset, a read offset, and an end offset, all of which are zero-based unsigned values, as follows:

- Write offset: The index in the dictionary where the next byte is added.
- Read offset: The index in the dictionary from which the next byte is read.
- End offset: The number of bytes currently in the dictionary. It MUST be less than or equal to 4096.

The end offset is incremented until its value is 4096.

2.1.3.1.5 Dictionary Reference

A dictionary reference is a 16-bit packed structure stored in the value of the **REFERENCE** field, as specified in section 2.1.3.1.1. The dictionary reference is stored in **big-endian** form on the wire. The format of this reference is as follows.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Offset												Length																			

Offset (12 bits): This field contains an index from the beginning of the dictionary that indicates where the matched content will start.

An offset that equals the write offset of the dictionary has the special meaning of completion of all compressed data, as specified in section 2.3.3.2, step 8. In this case, the writer MUST set the **Length** field to zero, and readers SHOULD ignore the **Length** field.

Length (4 bits): This value is 2 bytes less than the actual length of the content.

2.1.3.2 Calculate a CRC from a Given Array of Bytes

Given an initial value of the **CRC** field, as specified in section 2.1.3.1.1, or the value of the **CRC** field returned from a prior call (referred to in the following example as the **crcValue** field, which is a **DWORD** ([MS-DTYP])), the following is the algorithm for calculating the value of the **CRC** field for a given array of bytes (in pseudo-code).

```
FOR each byte in the input array
SET tablePosition to (crcValue XOR byte) BITWISE-AND 0xff
SET intermediateValue to crcValue RIGHTSHIFTED by 8 bits
SET crcValue to (crcTableValue at position tablePosition)
XOR intermediateValue
ENDFOR
RETURN crcValue
```

2.2 Decompression Algorithm Details

2.2.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this algorithm. The described organization is provided to facilitate the explanation of how the algorithm behaves. This document does not mandate that implementations adhere to this model, as long as their external behavior is consistent with that described in this document.

The abstract data model specified in section 2.1.1 also applies to decompression.

2.2.1.1 Input and Output

In this section, the input (the compressed RTF data, including the header) and the output (the decompressed data) are treated as streams.

2.2.2 Initialization

All initialization specified in section 2.1.2 is required by the decompression process, and therefore MUST be done.

2.2.2.1 Header

Before beginning decompression, the reader MUST read the **HEADER** field, as specified in section 2.1.3.1.1. If the value of the **COMPTYPE** field, as specified in section 2.1.3.1.1, is any value other than COMPRESSED or UNCOMPRESSED, then the reader MUST treat the input stream as corrupt.

If the value of the **COMPTYPE** field is COMPRESSED, then the reader MUST decompress the stream by using the compression algorithm specified in section 2.2.3.1.2. If the value of the **COMPTYPE** field is UNCOMPRESSED, then the contents are uncompressed and the reader MUST copy the contents as-is to the output stream, as specified in section 2.2.3.1.1.

2.2.2.2 Output

The output stream MUST initially have a length of zero.

2.2.3 Processing Rules

The reader MUST NOT validate the value of the **CRC** field when the **COMPTYPE** field is set to UNCOMPRESSED. The **CRC** and **COMPTYPE** fields are specified in section 2.1.3.1.1.

When the **COMPTYPE** field is set to COMPRESSED, the reader's decompression process MUST calculate the value of the **CRC** field for all of the **CONTENTS** fields, as specified in section 2.1.3.1.1, and compare that value to the value of the **CRC** field of the header. If the values do not match, then the reader MUST treat the input as corrupt.

If the decompression process, as defined in section 2.2, terminates prior to the end of the input, then the remainder of the input (the **PADDING** field, as specified in section 2.1.3.1.1,) MUST be included in the value of the **CRC** field. After this is done, if the computed value of the **CRC** field does not equal that which is specified in the **CRC** field of the header, then the reader MUST treat the input as corrupt.

2.2.3.1 Decompressing the Input

2.2.3.1.1 Decompressing Input of UNCOMPRESSED

The reader SHOULD read all bytes until the end of the stream is reached, regardless of the value of the **RAWSIZE** field. Or, the reader MAY read the number of bytes specified by the **RAWSIZE** field from the input (the **RAWDATA** field) and write them to the output. The **RAWSIZE** and **RAWDATA** fields are specified in section 2.1.3.1.1.

2.2.3.1.2 Decompressing Input of COMPRESSED

If at any point during the steps specified in this section, the end of the input is reached before the termination of decompression, then the reader MUST treat the input as corrupt.

The decompression process is a straightforward loop, as follows:

- Read the **CONTROL** field, as specified in section 2.1.3.1.1, from the input.
- Starting with the lowest bit (the 0x01 bit) in the **CONTROL** field, test each bit and carry out the actions as follows.
- After all bits in the **CONTROL** field have been tested, read another value of a **CONTROL** field from the input and repeat the bit-testing process.

For each bit, the reader MUST evaluate its value and complete the corresponding steps as specified in this section.

If the value of the bit is zero:

1. Read a 1-byte literal from the input and write it to the output.
2. Set the byte in the dictionary at the current write offset to the literal from step 1.
3. Increment the write offset and update the end offset, as appropriate, as specified in section 2.1.3.1.4.

If the value of the bit is 1:

1. Read a 16-bit dictionary reference from the input in big-endian byte-order.
2. Extract the offset from the dictionary reference, as specified in section 2.1.3.1.5.
3. Compare the offset to the dictionary's write offset. If they are equal, then the decompression is complete; exit the decompression loop.
4. Set the dictionary's read offset to offset.
5. Extract the length from the dictionary reference and calculate the actual length by adding 2 to the length that is extracted from the dictionary reference.
6. Read a byte from the current dictionary read offset and write it to the output.
7. Increment the read offset, wrapping as appropriate, as specified in section 2.1.3.1.4.
8. Write the byte to the dictionary at the write offset.
9. Increment the write offset and update the end offset, as appropriate, as specified in section 2.1.3.1.4.

10. Continue from step 6 until the number of bytes calculated in step 5 has been read from the dictionary.

The input value of the **CRC** field, as specified in section 2.1.3.1.1, MUST be calculated from every byte in the **CONTENTS** field, per the process specified in section 2.1.3.2. If the calculated value of the **CRC** field does not match the value of the **CRC** field in the header, then the reader MUST treat the input as corrupt.

2.3 Compression Algorithm Details

2.3.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this algorithm. The described organization is provided to facilitate the explanation of how the algorithm behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

The abstract data model specified in section 2.1.1 also applies to compression.

2.3.1.1 Input and Output

For the purpose of this section, the input (the uncompressed RTF data) and the output (the compressed data) will be treated as in-memory buffers of appropriate sizes. The output has an output cursor, which defines where the next byte of the output is written. The input has an input cursor, which defines the position from which the next byte of input is read.

2.3.1.2 Run Information

Compressing data when the value of the **COMPTYPE** field, as specified in section 2.1.3.1.1, is COMPRESSED is most easily understood and implemented if the writer does so one run at a time, writing each run to the output as it is completed. Information stored for a run includes:

- The current control byte (the **CONTROL** field, as specified in section 2.1.3.1.1) for the run, represented as a **BYTE** ([MS-DTYP]).
- A mask (called the control bit), represented as a **BYTE**.
- A token buffer, 16 bytes in length.
- The offset into the token buffer (the "token offset"), representing the next position in the buffer to which a token will be written.

In the implementation specified in the remainder of section 2.3, a run is considered completed when the value of the control bit is 0x80 after a token has been written.

2.3.2 Initialization

All initialization specified in section 2.1.2 is required by the compression process, and therefore MUST be done.

2.3.2.1 Input and Output

The writer MUST set the input cursor to the first byte in the input buffer.

The writer MUST set the output cursor to the 17th byte (to make space for the compressed header).

2.3.3 Processing Rules

When the **COMPTYPE** field is set to UNCOMPRESSED, the writer SHOULD NOT compute the value of the **CRC** field, and MUST set the value of the **CRC** field in the header to 0x00000000. The **COMPTYPE** and **CRC** fields are specified in section 2.1.3.1.1.

When the **COMPTYPE** field is set to COMPRESSED, the writer MUST calculate the value of the **CRC** field for every byte written to the **CONTENTS** field, as specified in section 2.1.3.1.1, and set the value of the **CRC** field of the header.

2.3.3.1 Compressing a Buffer of Uncompressed Contents with COMPTYPE UNCOMPRESSED

The writer MUST copy the uncompressed contents from the input buffer to the output buffer starting at the current output cursor. Compression MUST continue by filling in the header, as specified in section 2.3.3.1.1.

2.3.3.1.1 Filling in the Header

Using the fields defined in the RTF compression ABNF grammar, specified in section 2.1.3.1.1, the writer MUST fill in the header by using the following process:

1. Set the **COMPSIZE** field of the header to the length of the **CONTENTS** field in the output buffer plus 12.

2. Set the value of the **RAWSIZE** field of the header to the number of bytes read from the input.

3. Set the value of the **COMPTYPE** field of the header to UNCOMPRESSED.

4. Set the value of the **CRC** field of the header to zero.

2.3.3.2 Compressing a Buffer of Uncompressed Contents with COMPTYPE COMPRESSED

Compression proceeds as a loop, as follows:

1. The writer MUST (re)initialize the run by setting its control byte to zero, its control bit to 0x01, and its token offset to zero.

2. If there is no more input, then the writer MUST exit the compression loop (by advancing to step 8).

3. Locate the longest match in the dictionary for the current input cursor, as specified in section 2.3.3.2.1. Note that the dictionary is updated during this procedure.

4. If the match is zero or 1 byte in length, then the writer MUST copy the literal at the input cursor to the Run's token buffer at token offset. The writer MUST increment the token offset and the input cursor.

5. If the match is 2 bytes or longer, then the writer MUST create a dictionary reference, as specified in section 2.1.3.1.5, from the offset of the match and the length. (**Note**: The value stored in the **Length** field, as specified in section 2.1.3.1.5, is length minus 2). The writer MUST insert this dictionary reference in the token buffer as a big-endian word at the current token offset. The control bit MUST be bitwise ORed into the control byte, thus setting the bit that corresponds to the current token to 1. The writer MUST advance the token offset by 2 and MUST advance the input cursor by the length of the match.

6. If the control bit is not 0x80, then the control bit MUST be left-shifted by one bit and compression MUST continue building the run by returning to step 2.

7. If the control bit is equal to 0x80, then the writer MUST write the run to the output by writing the BYTE control byte, and then copying the token offset number of bytes from the token buffer to the output. The writer MUST advance the output cursor by the token offset plus 1 byte. Continue with compression by returning to step 1.

8. A dictionary reference MUST be created from an offset equal to the current write offset of the dictionary and a length of zero, and inserted in the token buffer as a big-endian word at the current token offset. The writer MUST then advance the token offset by 2. The control bit MUST be ORed into the control byte, thus setting the bit that corresponds to the current token to 1. When compressing zero bytes of data, the writer adds a null value during compression and the compressed run will be "02 00 0D 00" instead of "01 0C F0".

9. The writer MUST write the current run to the output by writing the value of the **CONTROL** field, as specified in section 2.1.3.1.1, and then copying the token offset number of bytes from the token buffer to the output. The output cursor is advanced by the token offset plus 1 byte.

After the output has been completed by execution of step 9, the writer MUST complete the output by filling the header, as specified in section 2.3.3.2.2.

2.3.3.2.1 Finding the Longest Match to Input

The purpose here is to scan over the dictionary to locate the longest string. It is important that, as the code finds a new longest match, the newly matched character SHOULD be added to the dictionary at that time (refer to the **AddByteToDictionary** procedure calls in the pseudo-code as follows).

In the case where the length of the match is zero, the literal that is being searched for MUST be added to the dictionary.

The scan MUST begin at the dictionary write offset plus 1 when the dictionary end offset is equal to 4096 bytes. When the end offset is less than 4096 bytes, the scan MUST begin at index zero. The scan SHOULD stop when 17 characters are matched but MUST stop after the finalOffset position is scanned, where finalOffset is defined as the dictionary write offset modulo 4096.

Matches that start at or before finalOffset and match across finalOffset allow a repeating sequence of characters, such as "XYZXYZXYZXYZ", to be represented as a series of appropriate initial literals ('X' 'Y' 'Z') and a single dictionary reference. (This example generates an offset of 210 and a length of 9, assuming that the dictionary is initialized as specified in section 2.1.2.1.) For a more detailed example, see section 3.2.2.

The longest match in the dictionary of the current position within the input can be computed by multiple implementation-dependent mechanisms. The following pseudocode is provided as one example; however, it is not necessary to follow this exactly, so long as the decompression algorithm specified in section 2.2 generates the original input given the compressed output generated.

```
PROCEDURE FindLongestMatch
SET finalOffset to the Write Offset of the Dictionary modulo 4096
IF the Dictionary's End Offset is not equal to the Dictionary buffer size THEN
SET matchOffset to 0
ELSE
SET matchOffset to (the Dictionary's Write Offset + 1) modulo 4096

ENDIF
```

```
SET bestMatchLength to 0

REPEAT
CALL TryMatch with matchOffset and the Input Cursor
SET matchOffset to (matchOffset + 1) modulo 4096
UNTIL matchOffset equals finalOffset
OR until bestMatchLength is 17 bytes long

IF bestMatchLength is 0 THEN
CALL AddByteToDictionary with the byte at Input Cursor
ENDIF
RETURN offset of bestMatchOffset and bestMatchLength
ENDPROCEDURE

PROCEDURE TryMatch
SET maxLength to the minimum of 17 and remaining bytes of Input
SET matchLength to 0
SET inputOffset to the Input Cursor
SET dictionaryOffset to matchOffset

WHILE matchLength is less than maxLength AND
the byte in the Dictionary at dictionaryOffset is equal to
the byte in Input at the inputOffset

INCREMENT matchLength
IF matchLength is greater than bestMatchLength THEN
CALL AddByteToDictionary with the byte
in Input at the inputOffset
ENDIF

INCREMENT inputOffset
SET dictionaryOffset to (dictionaryOffset + 1) modulo 4096
ENDWHILE

IF matchLength is greater than bestMatchLength THEN
SET bestMatchOffset to matchOffset
SET bestMatchLength to matchLength
ENDIF
ENDPROCEDURE

PROCEDURE AddByteToDictionary
SET the byte at the Dictionary's current Write Offset to the provided byte
IF the Dictionary's End Offset is less than the buffer size
THEN INCREMENT the End Offset
ENDIF
SET the Dictionary's Write Offset to
(the Dictionary's Write Offset + 1) modulo 4096
ENDPROCEDURE
```

2.3.3.2.2 Filling in the Header

Using the fields defined in the RTF compression ABNF grammar, as specified in section 2.1.3.1.1, the writer MUST fill in the header by using the following process:

1. Set the **COMPSIZE** field of the header to the number of **CONTENTS** bytes in the output buffer plus 12.

2. Set the **RAWSIZE** field of the header to the number of bytes read from the input.
3. Set the **COMPTYPE** field of the header to COMPRESSED.
4. Set the **CRC** field of the header to the value of the **CRC** field generated from the **CONTENTS** field.

3 Algorithm Examples

3.1 Decompressing Compressed RTF

In the following examples, the compressed RTF is examined in terms of "runs" for ease of exposition, where the term "run" refers to a control byte and the tokens that it represents. The length of a run can be computed from the control byte because each bit in the control byte that is set to zero represents a literal that is 1 byte long and each bit in the control byte that is set to 1 represents a dictionary reference that is 2 bytes long. Therefore, the length of a run (except the final run) is as follows.

```
run_length = 1 + (number of 0 bits) + (number of 1 bits) * 2
```

3.1.1 Example 1: Simple Compressed RTF

3.1.1.1 Compressed RTF Data

```
000000: 2d 00 00 00 2b 00 00 00-4c 5a 46 75 f1 c5 c7 a7
000010: 03 00 0a 00 72 63 70 67-31 32 35 42 32 0a f3 20
000020: 68 65 6c 09 00 20 62 77-05 b0 6c 64 7d 0a 80 0f
000030: a0
```

3.1.1.2 Compressed RTF Header

The first 16 bytes comprise the compressed RTF header.

```
000000: 2d 00 00 00 2b 00 00 00-4c 5a 46 75 f1 c5 c7 a7

COMPSIZE : 0x2d
RAWSIZE : 0x2b
COMPTYPE : COMPRESSED  ; 0x75465a4c
CRC: 0xa7c7c5f1
```

3.1.1.3 Initialization

The dictionary is initialized with the data, as described in section 2.1.2.1. After the initialization, the dictionary is as follows.

```
WritePosition: 207

      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123 4
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

3.1.1.4 Run 1

The first run begins on byte 16. The value of the **CONTROL** field, as described in section 2.1.3.1.1, at that location is 0x03. Represented as bits, the value of the **CONTROL** field would be %b00000011. The **CONTROL** field determines a run length, based on the number of '1' and '0' (zero) bits. Run length is equal to the number of '1' bits times 2 plus the number of '0' (zero) bits plus 1 for the **CONTROL** field itself. With a value of 0x3 for the **CONTROL** field, the run length is 11 bytes.

```
000010: 03 00 0a 00 72 63 70 67 31 32 35
```

Because the low-order bit in the **CONTROL** is a 1, the first token in the run is a dictionary reference and consists of the two bytes 00 and 0a. Reading these into a **WORD** data type ([MS-DTYP]) in big-endian order, the dictionary reference is 0x000a. As described in section 2.1.3.1.5, the offset into the dictionary is the upper 12 bits (for example, 0), and the length is the lower 4 bits (for example, 0xa). The length is stored as 2 less than the actual length, so 2 is added to the length, making the actual length 0x0C (12). Reading 12 bytes from the dictionary at offset zero returns the content "{\rtf1\ansi\".

```
      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
```

This content is copied to the output buffer and written to the write location for the dictionary. The new dictionary is as follows.

```
WritePosition: 219

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\

   Nonprintable characters:
       Position:0168   Byte:0x0d
       Position:0169   Byte:0x0a
```

The output stream is now as follows.

"{\rtf1\ansi\"

The next control bit is 1 (%b000000**1**1), specifying another dictionary reference for the bytes for which are 00 and 72. Converting to a **WORD** data type results in 0x0072, and extracting the offset and length results in offset equal to 0x0007, and a length of 0x4 (0x2+2).

Looking up the dictionary position 7 for 4 bytes results in: "ansi".

```
      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
```

This extracted content is appended to the output buffer and to the dictionary. The new dictionary is as follows.

```
WritePosition: 223

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansi

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The output stream is now as follows.

"{\rtf1\ansi\ansi"

The next control bit is 0 (%b00000**0**11), specifying a literal byte token. That token value is 0x63. Because it is a literal, no dictionary lookup happens. The byte is appended to the dictionary and to the output stream.

The new dictionary is as follows.

```
WritePosition: 224

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansic

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The output stream is now as follows.

"{\rtf1\ansi\ansic"

The next control bit is 0 (%b0000**0**011), specifying another literal byte token. That token value is 0x70. Because it is a literal, no dictionary lookup happens. The byte is appended to the dictionary and the output stream.

The new dictionary is as follows.

```
WritePosition: 225

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
```

```
0195: b\i\u\tab\tx{\rtf1\ansi\ansicp

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The output stream is now as follows.

"{\rtf1\ansi\ansicp"

Repeating for the remaining tokens in the run, the following bytes are added to the dictionary and the output stream (67 31 32 35).

The new dictionary is as follows.

```
WritePosition: 229

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansicpg125

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The output stream is now as follows.

"{\rtf1\ansi\ansicpg125"

The entire **CONTROL** field is now processed and the first run is now evaluated.

3.1.1.5 Run 2

The next run is now loaded and the same logic as described in Run 1 is executed.

RunSize: 11 bytes

```
00001b: 42 32 0a f3 20 68 65 6c 09 00 20
```

Bytes	Description
42	Control byte: 0x42 Bits: %b01000010
32	'2'
0a f3	Dictionary reference: 0af3 Offset: 0x0af (175) Length: [0x3+2] (50) Content: "\pard"

Bytes	Description
20	' '
68	'h'
65	'e'
6c	'l'
09 00	Dictionary reference: 0900 Offset: 0x090 (144) Length: [0x0+2] (2) Content: "lo"
20	' '

Dictionary:

```
WritePosition: 242

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansicpg1252\pard hello

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

OutputStream:

"{\rtf1\ansi\ansicpg1252\pard hello "

3.1.1.6 Run 3

The final run is 11 bytes, as follows.

```
000026: 62 77 05 b0 6c 64 7d 0a 80 0f a0
```

Bytes	Description
62	Control byte: 0x62 Bits: %b01100010
77	'w'
05 b0	Dictionary reference: 05b0 Offset: 0x05b (91) Length: [0x0+2] (2)

Bytes	Description
	Content: "or"
6c	'l'
64	'd'
7d	'}'
0a 80	Dictionary reference: 0a80 Offset: 0x0a8 (168) Length: [0x0+2] (2) Content: 0x0d 0x0a
0f a0	Dictionary reference: 0fa0 Offset: 0x0fa (250) Length: [0x0+2] (2) Content: <END>

The final dictionary reference is unique. The offset of 250 exactly matches the value of the **WritePosition** field at the time that the dictionary reference is encountered. This is an indicator that the end of the compressed content has been reached and decompression has to stop.

The final dictionary is as follows.

```
WritePosition: 250

      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123 4
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansicpg1252\pard hello world}__

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
 Position:0248 Byte:0x0d
 Position:0249 Byte:0x0a
```

The final decompressed output is as follows.

"{\rtf1\ansi\ansicpg1252\pard hello world}<CR><LF>"

3.1.2 Example 2: Reading a Token from the Dictionary that Crosses WritePosition

The following example illustrates that the requirement that bytes be added to the dictionary as they are copied to the output is necessary to allow longer matches than would otherwise be possible.

3.1.2.1 Compressed RTF

```
000000: 1a 00 00 00 1c 00 00 00-4c 5a 46 75 e2 d4 4b 51
000010: 41 00 04 20 57 58 59 5a-0d 6e 7d 01 0e b0
```

3.1.2.2 Compressed RTF Header

```
000000: 1a 00 00 00 1c 00 00 00-4c 5a 46 75 e2 d4 4b 51

COMPSIZE : 0x1a
RAWSIZE : 0x1c
COMPTYPE : COMPRESSED  ; 0x75465a4c
CRC: 0x514bd4e2
```

3.1.2.3 Initialization

The dictionary is initialized with the data, as described in section 2.1.2.1. After the initialization, the dictionary is as follows.

```
WritePosition: 207

      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123456789
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx

Nonprintable characters:
Position:0168 Byte:0x0d
Position:0169 Byte:0x0a
```

3.1.2.4 Run 1

The first run is 11 bytes long.

```
000010: 41 00 04 20 57 58 59 5a 0d 6e 7d
```

Bytes	Description
41	Control byte: 0x41 Bits: %b01000001
00 04	Dictionary reference: 0004 Offset: 0x000 (0) Length: [0x4+2](6) \Content: "{\rtf1"
20	' '
57	'W'
58	'X'
59	'Y'
5a	'Z'
0d 6e	Dictionary reference: 0d6e

Bytes	Description
	Offset: 0x0d6 (214) Length: [0x4+2](6) Content: "WXYZWXYZWXYZWXYZ"
7d	'}'

After the first dictionary reference and the first five literal tokens are processed, the dictionary is as follows.

```
WritePosition: 218

      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123 4
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1 WXYZ

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The output is now as follows.

"{\rtf1 WXYZ"

```
000018:0d 6e 7d
```

The next token in the input is a dictionary reference at offset 214 and length 16. There are only 4 bytes in the dictionary following that offset. As each byte of the dictionary reference is copied to the output, it is also added to the dictionary. Therefore, after the first four bytes of the dictionary reference are copied, the dictionary is as follows.

```
WritePosition: 222

      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123 4
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195b\i\u\tab\tx{\ref1 WXYZWXYZ

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The offset from which the dictionary reference is being copied has now been advanced from 214 to 218, which points to the newly written bytes, so the expansion continues with those bytes. The full expansion of the dictionary reference leads to a dictionary of the following.

```
WritePosition: 234
```

```
      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123 4
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195b\i\u\tab\tx{\ref1 WXYZWXYZWXYZWXYZWXYZ

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The output is as follows.

"{\ref1 WXYZWXYZWXYZWXYZWXYZ"

There is one more literal token in this run, as follows.

```
00001a: 7d
```

When decoded, this token leads to a dictionary of the following.

```
WritePosition: 235

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\ref1 WXYZWXYZWXYZWXYZWXYZ}

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The output is as follows.

"{\ref1 WXYZWXYZWXYZWXYZWXYZ}"

3.1.2.5 Run 2

This run is only 3 bytes, as follows.

```
00001b: 01 0e b0
```

Bytes	Description
01	Control byte: 0x01 Bits: %b00000001
0e b0	Dictionary reference: 0004 Offset: 0x0eb(235) Length: [0x0+2](2) Content: <END>

Because the offset of the dictionary reference is equal to the current value of the **WritePosition** field, this indicates that the decompression is complete.

3.2 Generating Compressed RTF

3.2.1 Example 1: Simple RTF

This example compresses the following RTF data.

"{\rtf1\ansi\ansicpg1252\pard hello world}<CR><LF>"

3.2.1.1 Initialization

The dictionary is initialized with the data, as described in section 2.3.2.1. After the initialization, the dictionary is as follows.

```
WritePosition: 207

      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123 4
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx
Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

CRC: 0

COMPSIZE: 0x000C

COMPTYPE: 0x75465a4c

The output is as follows.

```
Output cursor: 0x10

000000: 00 00 00 00 00 00 00 00-4c 5a 46 75 00 00 00 00
000010: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
000020: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
000030: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

InputCursor is: 0 (zero)

3.2.1.2 Run 1

Start by initializing the following run information.

```
Control byte: 0x00
Control bit:  0x01
Token offset: 0x00
Token buffer: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Input data is "{\rtf1\ansi\ansicpg1252\pard hello world}<CR><LF>".

The dictionary is now scanned starting at index zero, looping until through index 207, in an attempt to find the largest match of the input data.

The first match starts at position zero. As each new byte is matched, the byte is copied to the dictionary write index and the write index is incremented. This match stops at byte 12. The maximum length match is stored as 12 before moving to the next character. No larger match is found. Because the match is greater than one character, a dictionary reference has to be written to the output (length is encoded as match length minus 2). The dictionary reference written to the output is offset = 0, length = 10, 0x000A.

The **CONTROL** field, as described in section [2.1.3.1.1](), sets the value at the control bit set to 1 and advances the control bit to the next token.

The run information is now as follows.

```
Control byte: 0x01
Control bit:  0x02
Token offset: 0x02
Token buffer: 00 0a 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

The dictionary is now as follows.

```
WritePosition: 219

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The input data is now: "ansicpg1252\pard hello world}<CR><LF>".

Scanning the dictionary from index zero to index 219, new matches are calculated.

The first match is located at index 7. As each character is matched, it is moved to the dictionary write index. The match length is 4. No other larger match is located, and because the length is greater than one character, a dictionary reference is written to the output buffer (length is encoded as match length minus 2). The dictionary reference written to the output is offset = 7, length = 2, 0x0072.

The control bit location in the **CONTROL** field is set to 1, and the control bit is advanced.

The run information is now as follows.

```
Control byte: 0x03
Control bit:  0x04
Token offset: 0x04
Token buffer: 00 0a 00 72 00 00 00 00-00 00 00 00 00 00 00 00
```

The dictionary is now as follows.

```
WritePosition: 223

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansi

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The input data is now: "cpg1252\pard hello world}<CR><LF>".

Scanning the dictionary from index zero to index 223, new matches are located.

The first match is located at index 14. The 'c' character is matched, and is moved to the dictionary write index. The largest match is now 1 character. Continuing scanning, matches are located at positions 80 and 142, but because the match is not any larger, no additional characters are copied to the dictionary. Because the match is less than 2, a literal is written to the output stream.

The control bit location in the **CONTROL** field is set to zero and the control bit is advanced. The value of the **CONTROL** field is still 0x3 (%b00000011).

The run information is now as follows.

```
Control byte: 0x03
Control bit:  0x08
Token offset: 0x05
Token buffer: 00 0a 00 72 63 00 00 00-00 00 00 00 00 00 00 00
```

The dictionary is now as follows.

```
WritePosition: 224

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansic

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The input data is now: "pg1252\pard hello world}<CR><LF>".

Scanning the dictionary from index zero to index 224, new matches are located.

The first match is located at index 83. The 'p' character is matched, and is moved to the dictionary write index. The largest match is now 1 character. Continuing scanning, matches are located at positions 171, 176, and 181, but because the match is not any larger, no additional characters are copied to the dictionary. Because the match is less than 2, a literal is written to the output stream.

The control bit location in the **CONTROL** field is set to zero and the control bit is advanced.

The run information is now as follows.

```
Control byte: 0x03
Control bit:  0x10
Token offset: 0x06
Token buffer: 00 0a 00 72 63 70 00 00-00 00 00 00 00 00 00 00
```

The dictionary is now as follows.

```
WritePosition: 225

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansicp

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The input data is now: "g1252\pard hello world}<CR><LF>".

Scanning the dictionary from index zero to index 225, new matches are located.

The first match is located at index 156. The 'g' character is matched, and is moved to the dictionary write index. The largest match is now 1 character. Continuing scanning, matches are not found at any other locations. Because the match length is less than 2, a literal is written to the output stream.

The control bit location in the **CONTROL** field is set to zero and the control bit is advanced.

The run information is now as follows.

```
Control byte: 0x03
Control bit:  0x20
Token offset: 0x07
Token buffer: 00 0a 00 72 63 70 67 00-00 00 00 00 00 00 00 00
```

The dictionary is now as follows.

```
WritePosition: 226

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansicpg
```

```
Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The input data is now: "1252\pard hello world}<CR><LF>".

Scanning the dictionary from index zero to index 226, new matches are located.

The first match is located at index 5. The '1' character is matched, and is moved to the dictionary write index. The largest match is now 1 character. Continuing scanning, '1' also matches at 211, but the match length is still 1 character. Because the match length is less than 2, a literal is written to the output stream.

The control bit location in the **CONTROL** field is set to zero and the control bit is advanced.

The run information is now as follows.

```
Control byte: 0x03
Control bit:  0x40
Token offset: 0x08
Token buffer: 00 0a 00 72 63 70 67 31-00 00 00 00 00 00 00 00
```

The dictionary is now as follows.

```
WritePosition: 227

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansicpg1

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The input data is now: "252\pard hello world}<CR><LF>".

Scanning the dictionary from index zero to index 227, new matches are located.

The first match is located at index 29. The '2' character is matched, and is moved to the dictionary write index. The largest match is now 1 character. Continuing scanning, '2' also matches at 192, but the match length is still 1 character. Because the match length is less than 2, a literal is written to the output stream.

The control bit location in the **CONTROL** field is set to zero and the control bit is advanced.

The run information is now as follows.

```
Control byte: 0x03
Control bit:  0x80
Token offset: 0x09
```

```
Token buffer: 00 0a 00 72 63 70 67 31-32 00 00 00 00 00 00 00
```

The dictionary is now as follows.

```
WritePosition: 228

      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansicpg12

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The input data is now: "52\pard hello world}<CR><LF>".

Scanning the dictionary from index zero to index 228 for the character '5' results in zero matches.

Because the character is unmatched, it has to be moved to the dictionary write index. Because the match length is less than 2, a literal is also written to the output stream.

The control bit location in the **CONTROL** field is set to zero and the control bit is advanced.

In addition, because the control bit is now 0x80, it is not advanced; rather, the run is now written to the output.

The run information is now as follows.

```
Control byte: 0x03
Control bit:  0x80
Token offset: 0x0a
Token buffer: 00 0a 00 72 63 70 67 31-32 35 00 00 00 00 00 00
```

This is written to the output by writing the **CONTROL** field followed by token offset (0x0a) bytes from the token buffer. The output cursor is advanced by the number of bytes (0x0b) written to the output. The output is now as follows.

```
Output cursor: 0x1b

000000: 00 00 00 00 00 00 00 00-4c 5a 46 75 00 00 00 00
000010: 03 00 0a 00 72 63 70 67-31 32 35 00 00 00 00 00
000020: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
000030: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

This run is now complete.

3.2.1.3 Run 2

Prepare the next run by resetting the run information. The run information is now as follows.

```
Control byte: 0x00
Control bit:  0x01
Token offset: 0x00
Token buffer: 00 0a 00 72 63 70 67 31-32 00 00 00 00 00 00 00
```

Note that there is no need to overwrite data in the token buffer; that will be done as tokens are added.

```
Current Dictionary (WritePosition=229):
      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansicpg125

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

Input data is "2\pard hello world}<CR><LF>".

Add literal '2'; run information is as follows.

```
Control byte: 0x00
Control bit:  0x02
Token offset: 0x01
Token buffer: 32 0a 00 72 63 70 67 31-32 00 00 00 00 00 00 00
```

Input data is now "\pard hello world}<CR><LF>".

Add a dictionary reference (0x0af3) for the match of length 5 at offset 175 (matching "\pard"); the run information is as follows.

```
Control byte: 0x02
Control bit:  0x04
Token offset: 0x03
Token buffer: 32 0a f3 72 63 70 67 31-32 00 00 00 00 00 00 00
```

Input data is now " hello world}<CR><LF>".

Add literal ' '; the run information is as follows.

```
Control byte: 0x02
Control bit:  0x08
Token offset: 0x04
Token buffer: 32 0a f3 20 63 70 67 31-32 00 00 00 00 00 00 00
```

Input data is now "hello world}<CR><LF>".

Add a literal 'h'; the run information is as follows.

```
Control byte: 0x02
Control bit:  0x10
Token offset: 0x05
Token buffer: 32 0a f3 20 68 70 67 31-32 00 00 00 00 00 00 00
```

Input data is now "ello world}<CR><LF>".

Add a literal 'e'; the run information is as follows.

```
Control byte: 0x02
Control bit:  0x20
Token offset: 0x06
Token buffer: 32 0a f3 20 68 65 67 31-32 00 00 00 00 00 00 00
```

Input data is now "llo world}<CR><LF>".

Add literal 'l'; the run information is as follows.

```
Control byte: 0x02
Control bit:  0x40
Token offset: 0x07
Token buffer: 32 0a f3 20 68 65 6c 31-32 00 00 00 00 00 00 00
```

Input data is "lo world}<CR><LF>".

Add dictionary reference (0x0900) for a match of length 2 at offset 144 (matching "lo"); the run information is as follows.

```
Control byte: 0x42
Control bit:  0x80
Token offset: 0x09
Token buffer: 32 0a f3 20 68 65 6c 09-00 00 00 00 00 00 00 00
```

Input data is now " world}<CR><LF>".

Add literal ' '. Because the control bit is 0x80, the run is now complete. The run information is as follows.

```
Control byte: 0x42
Control bit:  0x80
Token offset: 0x0a
Token buffer: 32 0a f3 20 68 65 6c 09-00 20 00 00 00 00 00 00
```

Write the run to the output, which is now as follows.

```
Output cursor: 0x26

000000: 00 00 00 00 00 00 00 00-4c 5a 46 75 00 00 00 00
000010: 03 00 0a 00 72 63 70 67-31 32 35 42 32 0a f3 20
000020: 68 65 6c 09 00 20 00 00-00 00 00 00 00 00 00 00
000030: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

3.2.1.4 Run 3

Prepare the next run by resetting the run information. The run information is now as follows.

```
Control byte: 0x00
Control bit:  0x01
Token offset: 0x00
Token buffer: 32 0a f3 20 68 65 6c 09-00 20 00 00 00 00 00 00

Current Dictionary (WritePosition=242):
      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123 4
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1\ansi\ansicpg1252\pard hello

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

Input:"world}<CR><LF>"

Add literal 'w'; run information is as follows.

```
Control byte: 0x00
Control bit:  0x02
Token offset: 0x01
Token buffer: 77 0a f3 20 68 65 6c 09-00 20 00 00 00 00 00 00
```

Input:"orld}<CR><LF>"

Add dictionary reference (0x05b0) or match of length 2 at offset 91 (matching "or"); run information is as follows.

```
Control byte: 0x02
Control bit:  0x04
Token offset: 0x03
Token buffer: 77 05 b0 20 68 65 6c 09-00 20 00 00 00 00 00 00
```

Input: "ld}<CR><LF>"

Add literal 'l'; run information is as follows.

```
Control byte: 0x02
Control bit:  0x08
Token offset: 0x04
Token buffer: 77 05 b0 6c 68 65 6c 09-00 20 00 00 00 00 00 00
```

Input: "d}<CR><LF>"

Add literal 'd'; run information is as follows.

```
Control byte: 0x02
```

```
Control bit:  0x10
Token offset: 0x05
Token buffer: 77 05 b0 6c 64 65 6c 09-00 20 00 00 00 00 00 00
```

Input: "}<CR><LF>"

Add literal '}'; run information is as follows.

```
Control byte: 0x02
Control bit:  0x20
Token offset: 0x06
Token buffer: 77 05 b0 6c 64 7d 6c 09-00 20 00 00 00 00 00 00
```

Input: "<CR><LF>"

Add dictionary reference (0x0a80) for match of length 2 at offset 168 (matching "<CR><LF>"; run information is as follows.

```
Control byte: 0x22
Control bit:  0x40
Token offset: 0x08
Token buffer: 77 05 b0 6c 64 7d 0a 80-00 20 00 00 00 00 00 00
```

Input: <EMPTY>

Add a dictionary reference for termination. Because the dictionary's write cursor is 250, the reference is 0x0fa0. Run information is as follows.

```
Control byte: 0x62
Control bit:  0x80
Token offset: 0x0a
Token buffer: 77 05 b0 6c 64 7d 0a 80-0f a0 00 00 00 00 00 00
```

The run is now complete and is written to the output, as follows.

```
Output cursor: 0x31

000000: 00 00 00 00 00 00 00 00-4c 5a 46 75 00 00 00 00
000010: 03 00 0a 00 72 63 70 67-31 32 35 42 32 0a f3 20
000020: 68 65 6c 09 00 20 62 77-05 b0 6c 64 7d 0a 80 0f
000030: a0 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Having read through the input and written to the output, the header can now be filled in with the following:

RAWSIZE: 43

COMPSIZE: 45

CRC: 0xa7c7c5f1 (generated from bytes 0x0010 through 0x0030)

This results in the final output, as follows.

```
Output cursor: 0x31

000000: 2d 00 00 00 2b 00 00 00-4c 5a 46 75 f1 c5 c7 a7
000010: 03 00 0a 00 72 63 70 67-31 32 35 42 32 0a f3 20
000020: 68 65 6c 09 00 20 62 77-05 b0 6c 64 7d 0a 80 0f
000030: a0 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

The output is 0x031 bytes long.

3.2.2 Example 2: Compressing with Tokens that Cross WritePosition

This example compresses the following RTF data.

"{\rtf1 WXYZWXYZWXYZWXYZWXYZ}"

3.2.2.1 Initialization

The dictionary is initialized with the data, as described in section 2.3.2.1. After the initialization, the dictionary is as follows.

```
WritePosition: 207

      0         1         2         3         4         5         6
      0123456789012345678901234567890123456789012345678901234567890123 4
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

CRC: 0 (zero)

COMPSIZE: 0x000C

COMPTYPE: 0x75465a4c

Output is as follows.

```
Output cursor: 0x10

000000: 00 00 00 00 00 00 00 00-4c 5a 46 75 00 00 00 00
000010: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
000020: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
000030: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

InputCursor is: 0 (zero)

3.2.2.2 Run 1

Start by initializing the run information, as follows.

```
Control byte: 0x00
Control bit:  0x01
Token offset: 0x00
Token buffer: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Input data is "{\rtf1 WXYZWXYZWXYZWXYZWXYZ}".

Add a dictionary reference (0x0004) for a match of length 6 at offset zero (matching "{\rtf1"); run information is as follows.

```
Control byte: 0x01
Control bit:  0x02
Token offset: 0x02
Token buffer: 00 04 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Input data is now "WXYZWXYZWXYZWXYZWXYZ}".

Add literals ' ', 'W', 'X', 'Y', 'Z'; run information is as follows.

```
Control byte: 0x01
Control bit:  0x40
Token offset: 0x07
Token buffer: 00 04 20 57 58 59 5a 00-00 00 00 00 00 00 00 00
```

Input data is now "WXYZWXYZWXYZWXYZ}".

The dictionary is now as follows.

```
WritePosition: 218

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1 WXYZ

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

A match is found for the "WXYZ" at offset 214 in the dictionary, but because each character is added to the dictionary as it is matched, following the match of the initial 4 characters of the input, the dictionary is as follows.

```
WritePosition: 222

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1 WXYZWXYZ
```

```
Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

The match cursor of the input is now pointing at a 'W', as is the match cursor (at offset 218) of the dictionary. Therefore, matching continues, adding characters to the dictionary that can be matched later in the match. This terminates when a match of length 16 is found at offset 214 and the dictionary is as follows.

```
WritePosition: 234

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1 WXYZWXYZWXYZWXYZWXYZ

Nonprintable characters:
 Position:0168 Byte:0x0d
 Position:0169 Byte:0x0a
```

As a result, a dictionary reference (0x0d6e) is added for a length of 16 at offset 214 (matching " WXYZWXYZWXYZWXYZWXYZ"); run information is as follows.

```
Control byte: 0x41
Control bit:  0x80
Token offset: 0x09
Token buffer: 00 04 20 57 58 59 5a 0d-6e 00 00 00 00 00 00 00
```

Input data is now "}".

Add literal '}'; run information is as follows.

```
Control byte: 0x41
Control bit:  0x80
Token offset: 0x0a
Token buffer: 00 04 20 57 58 59 5a 0d-6e 7d 00 00 00 00 00 00
```

Because the control bit was 0x80, the run is written to the output, as follows.

```
Output cursor: 0x1b

000000: 00 00 00 00 00 00 00 00-4c 5a 46 75 00 00 00 00
000010: 41 00 04 20 57 58 59 5a-0d 6e 7d 00 00 00 00 00
000020: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
000030: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

3.2.2.3 Run 2

Start by initializing the run information, as follows.

```
Control byte: 0x00
Control bit:  0x01
Token offset: 0x00
Token buffer: 00 04 20 57 58 59 5a 0d-6e 7d 00 00 00 00 00 00
```

The dictionary is as follows.

```
WritePosition: 235

      0         1         2         3         4         5         6
      01234567890123456789012345678901234567890123456789012345678901234
0000: {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0065: ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0130: manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195: b\i\u\tab\tx{\rtf1 WXYZWXYZWXYZWXYZWXYZ}

Nonprintable characters:
Position:0168 Byte:0x0d
Position:0169 Byte:0x0a
```

Input data is <EMPTY>.

Because the input data is empty, a dictionary reference (0x0eb0) of length zero is added for the WritePosition; the run is as follows.

```
Control byte: 0x01
Control bit:  0x02
Token offset: 0x02
Token buffer: 0e b0 20 57 58 59 5a 0d-6e 7d 00 00 00 00 00 00
```

This is written to the output, as follows.

```
Output cursor: 0x1e

000000: 00 00 00 00 00 00 00 00-4c 5a 46 75 00 00 00 00
000010: 41 00 04 20 57 58 59 5a-0d 6e 7d 01 0e b0 00 00
000020: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
000030: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Finish by writing the header information:

RAWSIZE: 0x1a

COMPSIZE: 0x1c

CRC: 0x514bd4e2 (generated from bytes 0x0010 through 0x001d)

This results in the final output, as follows.

```
Output cursor: 0x1e

000000: 1a 00 00 00 1c 00 00 00-4c 5a 46 75 e2 d4 4b 51
000010: 41 00 04 20 57 58 59 5a-0d 6e 7d 01 0e b0 00 00
000020: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

```
000030: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

The output is 0x1e bytes long.

3.3 Generating the CRC

3.3.1 Example of CRC Generation

This example computes the value of the **CRC** field, as described in section 2.1.3.1.1, of the following bytes (the compressed input from section 3.1.1, with the header removed).

```
03 00 0a 00 72 63 70 67-31 32 35 42 32 0a f3 20
68 65 6c 09 00 20 62 77-05 b0 6c 64 7d 0a 80 0f
a0
```

The computation uses the procedure described in section 2.1.3.2.

3.3.1.1 Initialization

The **CRC** field, as described in section 2.1.3.1.1, is initially set to 0x00000000. The values in **crcTableValue** are also initialized, as described in section 2.1.2.2.1.

3.3.1.2 First Byte

The first byte is 0x03, and the current value of the **CRC** field, as described in section 2.1.3.1.1, is 0x00000000, so the **tablePosition** field is computed as follows.

```
tablePosition= (0x00000000 XOR 0x03) BITWISE-AND 0xff
= 0x00000003
```

This is used to index into the **crcTableValue** field, getting a table value of the following.

```
tableValue= 0x990951ba
```

The **intermediateValue** field is computed as follows.

```
intermediateValue= 0x00000000 RIGHTSHIFTED by 8 bits
= 0x00000000
```

The **CRC** field that incorporates this initial byte is now as follows.

```
CRC= 0x990951ba XOR 0x00000000
= 0x990951ba
```

3.3.1.3 Second Byte

The next byte is 0x00, and the current value for the **CRC** field, as described in section 2.1.3.1.1, is 0x990951ba, so the **tablePosition** field is computed as follows.

```
tablePosition= (0x990951ba XOR 0x00) BITWISE-AND 0xff
= 0xba
```

From which the **tableValue** field is as follows.

```
tableValue= 0x2bb45a92
```

The **intermediateValue** field is now as follows.

```
intermediateValue= 0x990951ba RIGHTSHIFTED by 8 bits
= 0x00990951
```

The updated **CRC** field is as follows.

```
CRC= 0x2bb45a92 XOR 0x00990951
= 0x2b2d53c3
```

3.3.1.4 Continuation

The computation proceeds as described, incorporating each byte into the value of the **CRC** field, as described in section 2.1.3.1.1.

The final value of the **CRC** field of this set of input bytes is 0xA7C7C5F1.

4 Security

4.1 Security Considerations for Implementers

Because the compressed content could originate from a malicious source, an implementer needs to be aware that certain sizes, such as COMPSIZE and RAWSIZE, might have been tampered with. Care needs to be taken to ensure that the client does not attempt to read or access data that is larger than the input during decompression. Few security risks exist during compression, as the algorithm can compress any content (not just RTF), and operates on the byte level.

4.2 Index of Security Parameters

None.

5 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Microsoft® Exchange Server 2003
- Microsoft® Exchange Server 2007
- Microsoft® Exchange Server 2010
- Microsoft® Exchange Server 15 Technical Preview
- Microsoft® Office Outlook® 2003
- Microsoft® Office Outlook® 2007
- Microsoft® Outlook® 2010
- Microsoft® Outlook® 15 Technical Preview

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.

6 Change Tracking

This section identifies changes that were made to the [MS-OXRTFCP] protocol document between the January 2012 and April 2012 releases. Changes are classified as New, Major, Minor, Editorial, or No change.

The revision class **New** means that a new document is being released.

The revision class **Major** means that the technical content in the document was significantly revised. Major changes affect protocol interoperability or implementation. Examples of major changes are:

- A document revision that incorporates changes to interoperability requirements or functionality.
- An extensive rewrite, addition, or deletion of major portions of content.
- The removal of a document from the documentation set.
- Changes made for template compliance.

The revision class **Minor** means that the meaning of the technical content was clarified. Minor changes do not affect protocol interoperability or implementation. Examples of minor changes are updates to clarify ambiguity at the sentence, paragraph, or table level.

The revision class **Editorial** means that the language and formatting in the technical content was changed. Editorial changes apply to grammatical, formatting, and style issues.

The revision class **No change** means that no new technical or language changes were introduced. The technical content of the document is identical to the last released version, but minor editorial and formatting changes, as well as updates to the header and footer information, and to the revision summary, may have been made.

Major and minor changes can be described further using the following change types:

- New content added.
- Content updated.
- Content removed.
- New product behavior note added.
- Product behavior note updated.
- Product behavior note removed.
- New protocol syntax added.
- Protocol syntax updated.
- Protocol syntax removed.
- New content added due to protocol revision.
- Content updated due to protocol revision.
- Content removed due to protocol revision.
- New protocol syntax added due to protocol revision.

- Protocol syntax updated due to protocol revision.
- Protocol syntax removed due to protocol revision.
- New content added for template compliance.
- Content updated for template compliance.
- Content removed for template compliance.
- Obsolete document removed.

Editorial changes are always classified with the change type **Editorially updated.**

Some important terms used in the change type descriptions are defined as follows:

- **Protocol syntax** refers to data elements (such as packets, structures, enumerations, and methods) as well as interfaces.
- **Protocol revision** refers to changes made to a protocol that affect the bits that are sent over the wire.

The changes made to this document are listed in the following table. For more information, please contact protocol@microsoft.com.

Section	Tracking number (if applicable) and description	Major change (Y or N)	Change type
2.2.3.1.1 Decompressing Input of UNCOMPRESSED	Removed product behavior note and moved information into the main text.	N	Product behavior note removed.
2.2.3.1.1 Decompressing Input of UNCOMPRESSED	Added normative language to distinguish between implemented product behavior and non-implemented optional behavior.	Y	Content updated.
2.3.3.2.1 Finding the Longest Match to Input	Removed a product behavior note and moved the information about being implementation-specific to the main text.	N	Product behavior note removed.

7 Index