

# [MS-OXRTFCP]: Rich Text Format (RTF) Compression Protocol Specification

## Intellectual Property Rights Notice for Protocol Documentation

- **Copyrights.** This protocol documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the protocols, and may distribute portions of it in your implementations of the protocols or your documentation as necessary to properly document the implementation. This permission also applies to any documents that are referenced in the protocol documentation.
- **No Trade Secrets.** Microsoft does not claim any trade secret rights in this documentation.
- **Patents.** Microsoft has patents that may cover your implementations of the protocols. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. However, the protocols may be covered by Microsoft's Open Specification Promise (available here: <http://www.microsoft.com/interop/osp/default.mspx>). If you would prefer a written license, or if the protocols are not covered by the OSP, patent licenses are available by contacting [protocol@microsoft.com](mailto:protocol@microsoft.com).
- **Trademarks.** The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights.

**Reservation of Rights.** All other rights are reserved, and this notice does not grant any rights other than specifically described above, whether by implication, estoppel, or otherwise.

**Preliminary Documentation.** This documentation is preliminary documentation for these protocols. Since the documentation may change between this preliminary version and the final version, 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.

**Tools.** This protocol documentation is intended for use in conjunction with publicly available standard specifications and networking programming art, and assumes that the reader is either familiar with the aforementioned material or has immediate access to it. A protocol specification does not require the use of Microsoft programming tools or programming environments in order for a Licensee to develop an implementation. Licensees who have access to Microsoft programming tools and environments are free to take advantage of them.

Revision Summary			
Author	Date	Version	Comments
Microsoft Corporation	April 4, 2008	0.1	Initial Availability.
Microsoft Corporation	April 25, 2008	0.2	Revised and updated property names and other technical content.

Preliminary

# Table of Contents

<b>1</b>	<b>Introduction</b> .....	<b>5</b>
1.1	Glossary.....	5
1.2	References.....	6
1.2.1	Normative References.....	6
1.2.2	Informative References.....	6
1.3	Protocol Overview (Synopsis).....	6
1.4	Relationship to Other Protocols.....	6
1.5	Prerequisites/Preconditions.....	6
1.6	Applicability Statement.....	6
1.7	Versioning and Capability Negotiation.....	7
1.8	Vendor-Extensible Fields.....	7
1.9	Standards Assignments.....	7
<b>2</b>	<b>Messages</b> .....	<b>7</b>
2.1	Transport.....	7
2.2	Message Syntax.....	7
2.2.1	RTF Compression Format.....	7
<b>3</b>	<b>Protocol Details</b> .....	<b>10</b>
3.1	Common Details.....	10
3.1.1	Abstract Data Model.....	10
3.1.2	Timers.....	11
3.1.3	Initialization.....	11
3.1.4	Higher-Layer Triggered Events.....	12
3.1.5	Message Processing Events and Sequencing Rules.....	13
3.1.6	Timer Events.....	13
3.1.7	Other Local Events.....	13
3.2	Decompression Details.....	13
3.2.1	Abstract Data Model.....	13
3.2.2	Timers.....	13
3.2.3	Initialization.....	13
3.2.4	Higher-Layer Triggered Events.....	14
3.2.5	Message Processing Events and Sequencing Rules.....	15
3.2.6	Timer Events.....	15
3.2.7	Other Local Events.....	15
3.3	Compression Details.....	15
3.3.1	Abstract Data Model.....	15
3.3.2	Timers.....	16
3.3.3	Initialization.....	16
3.3.4	Higher-Layer Triggered Events.....	16
3.3.5	Message Processing Events and Sequencing Rules.....	19
3.3.6	Timer Events.....	20
3.3.7	Other Local Events.....	20
<b>4</b>	<b>Protocol Examples</b> .....	<b>20</b>
4.1	Decompressing Compressed RTF.....	20

4.1.1	Example 1: Simple Compressed RTF .....	20
4.1.2	Example 2: Reading a Token from the Dictionary that Crosses WritePosition25 .....	25
4.2	Generating Compressed RTF .....	28
4.2.1	Example 1: Simple RTF .....	28
4.2.2	Example 2: Compressing with Tokens that Cross WritePosition .....	38
4.3	Generating the CRC .....	41
4.3.1	Example of CRC Generation .....	41
<b>5</b>	<b><i>Security</i></b> .....	<b>42</b>
5.1	Security Considerations for Implementers .....	42
5.2	Index of Security Parameters .....	42
<b>6</b>	<b><i>Appendix A: Office/Exchange Behavior</i></b> .....	<b>43</b>
<b>7</b>	<b><i>Index</i></b> .....	<b>44</b>

Preliminary

# 1 Introduction

Rich Text Format (RTF) (as specified in [MS-RTF]) is similar to Hypertext Markup Language (HTML) (as specified in [HTML4]) in that it can contain text and formatting information necessary to describe and render formatting and content. It can also contain references to other data such as fields, hyperlinks, and other RTF objects. Like HTML, RTF contains a reasonable amount of repeated content; thus it is desirable to compress RTF in order to reduce bytes over the wire.

The RTF Compression Protocol specifies:

- How to serialize raw RTF into a compressed format.
- How to serialize raw RTF in an uncompressed format.
- How to extract raw RTF from serialized content.

## 1.1 Glossary

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

**Augmented Backus-Naur Form (ABNF)**  
**HTML**  
**little-endian**  
**Rich Text Format (RTF)**

The following data types are defined in [MS-DTYP]:

**BYTE**  
**DWORD**  
**OCTET**  
**WORD**

The following terms are specific to this document:

**big-endian:** Multiple-byte values that are byte-ordered with the most significant byte stored in the memory location with the lowest address.

**CRC:** See **Cyclical Redundancy Check**.

**Cyclical Redundancy Check:** A computable value which can be used to validate content when sent over the wire or decompressed.

**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***

### **1.2.1 Normative References**

[MS-DTYP] Microsoft Corporation, "Windows Data Types", March 2007, <http://go.microsoft.com/fwlink/?LinkId=111558>.

[MS-OXGLOS] Microsoft Corporation, "Office Exchange Protocols Master Glossary", April 2008.

[MS-OXPROPS] Microsoft Corporation, "Office Exchange Protocols Master Property List Specification", April 2008.

[MS-RTF] Microsoft Corporation, "Word 2007: Rich Text Format (RTF) Specification, Version 1.9", February 2007, <http://go.microsoft.com/fwlink/?LinkId=112393>.

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

[RFC5234] Crocker, D., Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 5234, January 2008, <http://www.ietf.org/rfc/rfc5234.txt>.

### **1.2.2 Informative References**

[HTML401] World Wide Web Consortium, "HTML 4.01 Specification", December 1999, <http://www.w3.org/TR/html401/>.

## ***1.3 Protocol Overview (Synopsis)***

This document covers the mechanism for compressing and decompressing RTF.

## ***1.4 Relationship to Other Protocols***

The RTF Compression Protocol requires no additional protocols to accomplish the specified work. The PidTagRtfCompressed property (as specified in [MS-OXPROPS] and [MS-OXCMSG]) relies on this protocol.

## ***1.5 Prerequisites/Preconditions***

None.

## ***1.6 Applicability Statement***

This protocol is specifically used with information from the message object's PidTagRtfCompressed property. Clients that do not implement this protocol will be unable to interpret the data which was packed with this protocol. This protocol can be used to compress

6 of 44

and decompress any content. Additionally, this protocol supports storing content in an uncompressed form.

### **1.7 Versioning and Capability Negotiation**

None.

### **1.8 Vendor-Extensible Fields**

None.

### **1.9 Standards Assignments**

None.

## **2 Messages**

### **2.1 Transport**

None.

### **2.2 Message Syntax**

#### **2.2.1 RTF Compression Format**

Unless otherwise specified, sizes in this section are expressed in BYTES and multiple-byte values are stored in little-endian format.

##### **2.2.1.1 RTF Compression ABNF Grammar**

This section defines the format of the contents stored in the PidTagRtfCompressed property.

RTFCOMPRESSED = HEADER CONTENTS

; The size of the HEADER is sixteen (0x0010) bytes

HEADER = COMPSIZE RAWSIZE COMPTYPE CRC

; Clients MUST set to the length of the compressed data (CONTENTS)

; in bytes plus the count of the remaining bytes from HEADER

; (0x0010 – 0x0004 = 0x000C).

COMPSIZE = DWORD

; Size in bytes of the uncompressed content

RAWSIZE = DWORD

; Type of Compression

COMPTYPE = COMPRESSED / UNCOMPRESSED

COMPRESSED = %x4C.5A.46.75 ; 0x75465A4C

UNCOMPRESSED = %x4D.45.4C.41 ; 0x414C454D

[MS-OXRTFCP] - v0.2

Rich Text Format (RTF) Compression Protocol Specification

Copyright © 2008 Microsoft Corporation.

Release: Friday, April 25, 2008

; If COMPTYPE is COMPRESSED then the cyclical redundancy check computed from  
 ; the CONTENTS.

; If the COMPTYPE is UNCOMPRESSED then the CRC MUST be %x00.00.00.00

CRC = DWORD  
 CONTENTS = RAWDATA / COMPRESSEDATA

; If COMPTYPE is UNCOMPRESSED

RAWDATA = \*LITERAL

; If COMPTYPE is COMPRESSED

COMPRESSEDATA = [\*RUN] ENDRUN [PADDING]

RUN = CONTROL 8\*8TOKEN

ENDRUN = CONTROL 1\*8TOKEN

CONTROL = OCTET

TOKEN = REFERENCE / LITERAL

REFERENCE = WORD ; big-endian

LITERAL = OCTET

PADDING = \*OCTET

### 2.2.1.2 Compressed RTF

The content of compressed RTF consists of a header and a series of runs. The number of runs will vary based on the quantity of content being compressed and sizes of the matches in the dictionary.

HEADER	RUN <sub>1</sub>	RUN <sub>2</sub>	RUN <sub>3</sub>
RUN <sub>4</sub>	...	ENDRUN	PADDING

The ABNF grammar specified in section 2.2.1.1 contains necessary details supplementary to the constructs defined in this section.

### 2.2.1.3 Compressed Run

A run (RUN) is composed of a Control Byte (CONTROL) and eight (8) variable sized tokens. The final run (ENDRUN) can contain fewer than eight (8) tokens.

CONTROL	TOKEN <sub>1</sub>	TOKEN <sub>2</sub>	TOKEN <sub>3</sub>	TOKEN <sub>4</sub>	TOKEN <sub>5</sub>	TOKEN <sub>6</sub>	TOKEN <sub>7</sub>	TOKEN <sub>8</sub>
1 Byte	Varies	Varies	Varies	Varies	Varies	Varies	Varies	Varies



Tokens are either a dictionary reference (see section 2.2.1.5) or literals, depending on the value of the corresponding bit in the Control Byte.

### *Control Byte*

Each Control Byte (CONTROL) contains information on how to interpret the next eight (8) tokens. The low bit (bitmask %x1) the CONTROL corresponds to Token1, the second bit (bitmask %x2) corresponds to Token2, and so forth. In ENDRUN, the bits in CONTROL after the completion dictionary reference (see section 2.2.1.5) are undefined and MUST be ignored.

### *Token Semantics*

The type of token and its meaning depend on the value of the corresponding bit in the CONTROL:

- If the bit in the CONTROL is zero (0), the corresponding token is a one-byte literal representing the exact byte in the uncompressed content.
- If the bit in the CONTROL is one (1), the corresponding token is a two-byte dictionary reference that indicates the offset and length of a series of bytes in the dictionary corresponding to the bytes in the uncompressed content. (See section 2.2.1.5 for more information.)

#### **2.2.1.4 Dictionary**

This protocol uses a dictionary which 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 first byte and 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.

- write offset: the index in the dictionary where the next byte will be added.
- read offset: the index in the dictionary from which the next byte will be read.
- end offset: the number of bytes currently in the dictionary, it MUST be less than or equal to 4096.

The end offset will be incremented until its value is 4096.

#### **2.2.1.5 Dictionary Reference**

A dictionary reference is a sixteen bit packed structure stored in REFERENCE. The dictionary reference is stored in big-endian form on the wire. The format of this reference is:

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

Length is comprised of the lowest four (4) bits of the dictionary reference. The length is stored as two (2) fewer than the actual length.

Offset is comprised of the upper twelve (12) bits of the dictionary reference. The offset is an index from the beginning of the dictionary indicating 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 (see section 3.3.4.2, step 8). The writer **MUST** set the length to 0 in this case. Readers **SHOULD** ignore the length specified.

## **3 Protocol Details**

### **3.1 Common Details**

#### **3.1.1 Abstract Data Model**

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol 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.

##### **3.1.1.1 CRC Information**

The client uses a 32-bit Cyclical Redundancy Check (CRC) stored in the **HEADER** of **RTFCOMPRESSED** to ensure the validity of the compressed contents during decompression. During compression, the client generates the CRC of the compressed contents.

A pre-computed table of values is used for the CRC generation (see section 3.1.3.2.1).

##### **3.1.1.1.1 Decompression**

The client **MUST NOT** validate the CRC when **COMPTYPE** is **UNCOMPRESSED**.

When **COMPTYPE** is **COMPRESSED**, the client's decompression process **MUST** calculate the CRC for all of **CONTENTS** and compare that value to the value of the CRC field of the **HEADER**. If the values do not match, the client **MUST** treat the input as corrupt.

If the decompression process (as defined in section 3.2) terminates prior to the end of the input, the remainder of the input (**PADDING**) **MUST** be included in the CRC. Once this is done, if the computed CRC does not equal that specified in the CRC field of the **HEADER**, the client **MUST** treat the input as corrupt.

##### **3.1.1.1.2 Compression**

When **COMPTYPE** is **UNCOMPRESSED**, the client **SHOULD NOT** compute the CRC, and **MUST** set the CRC field in the **HEADER** to zero (0).

When **COMPTYPE** is **COMPRESSED**, the client **MUST** calculate the CRC for every byte written to **CONTENTS** and set the value of the CRC field of the **HEADER**.

## 3.1.2 Timers

None.

## 3.1.3 Initialization

### 3.1.3.1 Dictionary

The client MUST initialize the dictionary (starting at offset 0) with the ASCII string:

```
{\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil<SP>\froman<SP>\fswiss<SP>\fmodern<SP>\fscript<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)

Once the dictionary is initialized, the client MUST set the write offset and the end offset of the dictionary to 207 (pointing to the byte following the pre-loaded string).

NOTE: The Dictionary will not be used when COMPTYPE is UNCOMPRESSED.

### 3.1.3.2 CRC

The client MUST initialize the CRC to 0.

#### 3.1.3.2.1 CRC Lookup Table

The pre-computed table used for CRC generation MUST contain the following 256 DWORDS:

```
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, 0xbf061116,  
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, 0x7edcfe77, 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, 0xcd70693, 0x54de5729, 0x23d967bf,  
0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,  
0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d

### 3.1.4 Higher-Layer Triggered Events

#### 3.1.4.1 Calculate a CRC from a given array of bytes.

Given an initial CRC or the CRC returned from a prior call (referred to below as `crcValue`, which is a `DWORD`), the algorithm for calculating the CRC of a given array of bytes is (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

### **3.1.5 Message Processing Events and Sequencing Rules**

None.

### **3.1.6 Timer Events**

None.

### **3.1.7 Other Local Events**

None.

## **3.2 Decompression Details**

### **3.2.1 Abstract Data Model**

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol 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 3.1.1 also applies to decompression.

#### **3.2.1.1 Input and Output**

For purposes of this section, the input (the compressed RTF data, including the HEADER) and the output (the decompressed data) will be treated as streams.

#### **3.2.2 Timers**

None.

#### **3.2.3 Initialization**

All initialization specified in section 3.1.3 is required by the decompression process and therefore **MUST** be done.

##### **3.2.3.1 Header**

Before beginning decompression, the client **MUST** read the HEADER (as specified in section 2.2.1.1). If COMPTYPE is any value other than COMPRESSED or UNCOMPRESSED, the client **MUST** treat the input stream as corrupt.

If COMPTYPE is COMPRESSED, the client **MUST** decompress the stream using the compression algorithm specified in section 3.2.4.1.2. If COMPTYPE is UNCOMPRESSED, the contents are uncompressed and the client **MUST** copy the contents as-is to the output stream, as specified in section 3.2.4.1.1.

### 3.2.3.2 Output

The output stream **MUST** initially have a length of 0.

## 3.2.4 Higher-Layer Triggered Events

### 3.2.4.1 Decompressing the Input

#### 3.2.4.1.1 Decompressing Input of UNCOMPRESSED

The client **SHOULD** read RAWSIZE bytes (as specified in section 2.2.1.1) from the input (RAWDATA) and write them to the output <1>.

#### 3.2.4.1.2 Decompressing Input of COMPRESSED

If at any point during the steps specified below, the end of the input is reached before the termination of decompression, the client **MUST** treat the input as corrupt.

The decompression process is a straightforward loop:

- Read a CONTROL from the input.
- Starting with the lowest bit (the 0x01 bit) in the CONTROL, test each bit and carry out the actions specified below.
- Once all bits in the CONTROL have been tested, read another CONTROL from the input and repeat the bit testing process.

For each bit, the client **MUST** evaluate its value and complete the corresponding steps as specified below.

If the bit value is 0:

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 (see section 2.2.1.4).

If the bit value 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 (see section 2.2.1.5).
3. Compare offset to the dictionary's write offset. If they are equal, the decompression is complete; exit the decompression loop.
4. Set the dictionary's read offset to offset.
5. Extract length from the dictionary reference (see section 2.2.1.5).
6. Read a byte from the current dictionary read offset and write it to the output.
7. Increment the read offset, wrapping as appropriate (see section 2.2.1.4).
8. Write the byte to the dictionary at the write offset.

9. Increment the write offset and update the end offset, as appropriate (see section 2.2.1.4).

10. Continue from step (6) until length bytes have been read from the dictionary.

The input CRC MUST be calculated from every byte in CONTENT, per the process specified in section 3.1.4.1. If the calculated CRC does not match the CRC field in the HEADER, the client MUST treat the input as corrupt.

### **3.2.5 Message Processing Events and Sequencing Rules**

None.

### **3.2.6 Timer Events**

None.

### **3.2.7 Other Local Events**

None.

## **3.3 Compression Details**

### **3.3.1 Abstract Data Model**

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol 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 3.1.1 also applies to compression.

#### **3.3.1.1 Input and Output**

For purposes 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 to be written; the input has an input cursor, which defines the position from which the next byte of input is to be read.

#### **3.3.1.2 Run Information**

Compressing data with COMPTYPE COMPRESSED is most easily understood and implemented if the client does so one run at a time, writing each run to the output as it is completed. Information to be stored for a run includes:

- The current control byte (CONTROL) for the run, represented as a BYTE.
- A mask (called the control bit), represented as a BYTE.
- A token buffer of 16 BYTES.
- The offset into the token buffer (“token offset”), representing the next position in the buffer to which a token will be written.

In the implementation outlined in the remainder of section 3.3, a run is considered "completed" if the value of the control bit is 0x80 after a token has been written.

### **3.3.2 Timers**

None.

### **3.3.3 Initialization**

All initialization specified in section 3.1.3 is required by the compression process and therefore MUST be done.

#### **3.3.3.1 Input and Output**

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

The client MUST set the output cursor to the 17<sup>th</sup> byte (to make space for the compressed HEADER).

### **3.3.4 Higher-Layer Triggered Events**

#### **3.3.4.1 Compressing a Buffer of Uncompressed Contents with COMPTYPE UNCOMPRESSED**

The client 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 3.3.4.1.1).

##### **3.3.4.1.1 Filling in the Header**

The client MUST fill in the HEADER by the following process:

1. The COMPSIZE (see section 2.2.1.1) field of the HEADER MUST be set to the number of Content bytes in Output Buffer plus 12.
2. The RAWSIZE (see section 2.2.1.1) field of the HEADER MUST be set to the number of bytes read from the input.
3. The COMPTYPE (see section 2.2.1.1) field of the HEADER MUST be set to UNCOMPRESSED.
4. The CRC (see section 2.2.1.1) field of the HEADER MUST be set to zero (0).

#### **3.3.4.2 Compressing a Buffer of Uncompressed Contents with COMPTYPE COMPRESSED**

Compression proceeds as a loop:

1. The client MUST (re)initialize the run by setting its Control Byte to 0, its Control Bit to 0x01, and its Token Offset to 0.
2. If there is no more input, the client MUST exit the compression loop (by advancing to step (8) below).
3. Locate the longest match in the Dictionary for the current input cursor as specified in section 3.3.4.2.1. Note that the dictionary is updated during this procedure.



4. If the match is 0 or 1 byte in length, the client MUST copy the literal at the input cursor to the Run's Token Buffer at token offset. The client MUST increment the token offset and the input cursor.
5. If the match is 2 bytes or longer, the client MUST create a dictionary reference (see section 2.2.1.5) from the offset of the match and the length (note: the value stored in REFERENCE is length minus 2). The client 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 OR-ed into the Control Byte, thus setting the bit corresponding to the current token to 1. The client 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, 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, the client MUST write the run to the output by writing the BYTE Control Byte, and then copying Token Offset number of BYTES from the token buffer to the output. The client MUST advance the output cursor by Token Offset + 1 BYTES. Continue with compression by returning to step (1).
8. A dictionary reference (see section 2.2.1.5) MUST be created from an offset equal to the current write offset of the dictionary and a length of 0, and inserted in the token buffer as a big-endian word at the current token offset. The client MUST then advance the token offset by 2. The Control Bit MUST be OR-ed into the Control Byte, thus setting the bit corresponding to the current token to 1 <2>.
9. The client MUST write the current run to the output by writing the BYTE Control Byte, and then copying Token Offset number of BYTES from the token buffer to the output. The output cursor is advanced by Token Offset + 1 BYTES.

Once the output has been completed by execution of step (9), the client MUST complete the output by filling the HEADER as specified in section 3.3.4.2.2.

#### 3.3.4.2.1 Finding the Longest Match to Input

The purpose here is to scan over the dictionary locating 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 calls in the pseudocode below).

In the case where the length of the match is zero (0), the literal 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 (0). The scan SHOULD stop when 17 characters are matched but MUST stop once the *finalOffset* position is scanned.

Matches which 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 will

generate an offset of 210, and a length of 9, assuming the dictionary was initialized as specified in section 3.1.3.1). Refer to section 4.2.2 for a more detailed example.

The longest match in the dictionary of the current position within the input MAY <3> be computed by the following pseudo-code. It is not necessary to follow this exactly, so long as the decompression algorithm specified in section 3.2 will generate 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

    ENDF
    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
    ENDF
    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

```

#### **3.3.4.2.2 Filling in the Header**

The client **MUST** fill in the Header by the following process:

1. The COMPSIZE (see section 2.2.1.1) field of the HEADER **MUST** be set to the number of CONTENT bytes in Output Buffer plus 12.
2. The RAWSIZE (see section 2.2.1.1) field of the HEADER **MUST** be set to the number of bytes read from the input.
3. The COMPTYPE (see section 2.2.1.1) field of the HEADER **MUST** be set to COMPRESSED.
4. The CRC (see section 3.1.3.2) field of the HEADER **MUST** be set to the CRC (see section 3.1.1.1.2) generated from the CONTENT bytes.

### **3.3.5 Message Processing Events and Sequencing Rules**

None.

### 3.3.6 Timer Events

None.

### 3.3.7 Other Local Events

None.

## 4 Protocol Examples

### 4.1 Decompressing Compressed RTF

In the following examples, the Compressed RTF will be 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 0 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. Thus, the length of a run is:

$$\text{run\_length} = 1 + (\text{number of 0 bits}) + (\text{number of 1 bits}) * 2$$

#### 4.1.1 Example 1: Simple Compressed RTF

##### 4.1.1.1 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
```

##### 4.1.1.2 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
```

##### 4.1.1.3 Initialization

The Dictionary is initialized with the data as described in section 2.2.1.4. After the initialization the Dictionary is:

```
WritePosition: 207
```

```
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
```

NonPrintable Characters:

Position:0168      Byte:0x0d  
Position:0169      Byte:0x0a

#### 4.1.1.4 Run 1

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

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

Since 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 0a. Reading these into a WORD in big-endian order the dictionary reference is 0x000a. As specified in section 2.2.1.5, the offset into the dictionary is the upper 12 bits (i.e. 0), and the length is the lower 4 bits (i.e. 0xa). The length is stored 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 0 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:

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:

```
“{\rtf1\ansi\
```

The next control bit is 1 (%b00000011), specifying another dictionary reference whose bytes are 00 72. Converted to a WORD gives us 0x0072, and extracting the offset and length results in offset=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:

WritePosition: 223

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

```
NonPrintable Characters:
      Position:0168      Byte:0x0d
      Position:0169      Byte:0x0a
```

The output stream is now:

```
“{\rtf1\ansi\ansi”
```

The next control bit is 0 (%b00000011), specifying a literal byte token. That token value is 0x63. Since it is a literal, no dictionary lookup happens. The byte is appended to the dictionary and the output stream.

The new dictionary is:

WritePosition: 224

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

```
NonPrintable Characters:
      Position:0168      Byte:0x0d
      Position:0169      Byte:0x0a
```

The output stream is now:

```
“{\rtf1\ansi\ansic”
```

The next control bit is 0 (%b00000011), 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:

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:

“{\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:

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:

“{\rtf1\ansi\ansicpg125”

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

#### 4.1.1.5 Run 2

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

RunSize: 11 Bytes

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

Control Byte: 0x42 bits: %b01000010

```
32      '2'  
0a f3 Dictionary Ref:0af3  
      Offset: 0x0af(175) Length: [0x3+2] (50)  
      Content:"\pard"  
20      ' '  
68      'h'  
65      'e'  
6c      'l'  
09 00 Dictionary Ref:0900  
      Offset: 0x090(144) Length: [0x0+2] (2)  
      Content:"lo"  
20      ' '
```

Dictionary:

WritePosition: 0242

```
      0          1          2          3          4          5          6  
01234567890123456789012345678901234567890123456789012345678901234  
0000  {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnl1 \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 “

### 4.1.1.6 Run 3

The final run is 11 Bytes.

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



Control Byte: 0x62 bits: %b01100010

```
77      'w'
05 b0 Dictionary Ref:05b0
      Offset: 0x05b(91) Length: [0x0+2] (2)
      Content:"or"
6c      'l'
64      'd'
7d      '}'
0a 80 Dictionary Ref:0a80
      Offset: 0x0a8(168) Length: [0x0+2] (2)
      Content:0x0d 0x0a
0f a0 Dictionary Ref:0fa0
      Offset: 0x0fa(250) Length: [0x0+2] (2)
      Content: <END>
```

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

The final dictionary is:

WritePosition: 250

```
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 world}__
```

NonPrintable Characters:

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

The final decompressed output is:

“{\rtf1\ansi\ansicpg1252\pard hello world}<CRLF>”

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

The following example shows 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.

### 4.1.2.1 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
```

### 4.1.2.2 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
```

### 4.1.2.3 Initialization

The dictionary is initialized with the data as described in section 3.1.3.1. After the initialization the dictionary is:

WritePosition: 207

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

```
NonPrintable Characters:
      Position:0168      Byte:0x0d
      Position:0169      Byte:0x0a
```

### 4.1.2.4 Run 1

The first run is 11 bytes long:

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

```
Control Byte: 0x41      bits: %b01000001
00 04      Dictionary Ref:0004
      Offset: 0x000(0) Length: [0x4+2] (6)
      Content:"{\rtf1"
20          ' '
57          'W'
58          'X'
59          'Y'
5a          'Z'
0d 6e      Dictionary Ref:0d6e
      Offset: 0x0d6(214) Length: [0xe+2] (16)
      Content:"WXYZWXYZWXYZWXYZ"
7d          '}'
```

After the first dictionary reference and the first five literal tokens are processed, the dictionary is:

WritePosition: 218

```
      0          1          2          3          4          5          6
0000  01234567890123456789012345678901234567890123456789012345678901234
0065  {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
      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 at this point is

```
"{\rtf1 WXYZ"
```

```
0018: 0d 6e 7d
```

The next token in the input is a dictionary reference at offset 214 and of 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. Thus, after the first four bytes of the dictionary reference are copied, the dictionary is:

```
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{\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:

```
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{\ref1 WXYZWXYZWXYZWXYZWXYZWXYZ
```

```
NonPrintable Characters:  
    Position:0168      Byte:0x0d  
    Position:0169      Byte:0x0a
```

and output of:

```
"{\ref1 WXYZWXYZWXYZWXYZWXYZWXYZ"
```

There is one more literal token in this run:

```
0001a: 7d
```

Which when decoded leads to a dictionary of:

```
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

```

and output of:

```
"{\ref1 WXYZWXYZWXYZWXYZWXYZ}"
```

#### 4.1.2.5 Run 2

This run is only 3 bytes:

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

```

Control Byte: 0x1 bits: %b00000001
      0e b0      Dictionary Ref:0eb0
                  Offset: 0x0eb(235) Length: [0x0+2] (2)
                  Content: <END>

```

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

## 4.2 Generating Compressed RTF

### 4.2.1 Example 1: Simple RTF

This example will compress the following RTF data.

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

#### 4.2.1.1 Initialization

The dictionary is initialized with the data as described in section 3.3.3.1. After the initialization the dictionary is:

```
WritePosition: 207
```

```

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
NonPrintable Characters:
      Position:0168      Byte:0x0d
      Position:0169      Byte:0x0a

```

CRC is: 0

COMPSIZE is: 0x000C

COMPTYPE is: 0x75465a4c

Output is:

Output Cursor: 0x10

```

000000: 00 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

### 4.2.1.2 Run 1

Start by initializing the 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 0, looping until through index 207, attempting to find the largest match of the input data.

The first match starts at position 0. 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 max length match is stored as 12, before moving to the next character. No larger match is found. Since the match is greater than 1 character a dictionary reference has to be written to the output (length is encoded as match length – 2).

Dictionary reference contents, Offset = 0, Length = 10, 0x000A

The CONTROL sets the value at the control bit set to 1 and advances the control bit to the next token.

The run information at this point is:

```

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:

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 0 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 since the length is greater than 1 character, a dictionary reference is written to the output buffer (length is encoded as match length – 2).

Dictionary reference contents, Offset = 7, Length = 2, 0x0072

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

The run information at this point is:

```
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:

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 0 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. Since the match is less than 2, a literal is written to the output stream.

The control bit location in the CONTROL is set to 0 and the control bit is advanced. The CONTROL is still 0x3 [%b00000011].

The run information at this point is:

```
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
```

The dictionary is now:

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 0 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. Since the match is less than 2, a literal is written to the output stream.

The control bit location in the CONTROL is set to 0 and the control bit is advanced.

The run information at this point is:

```
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
```

The dictionary is now:

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 0 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 is set to 0 and the control bit is advanced.

The run information at this point is:

```
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:

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 0 to index 226, new matches are located.

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

The control bit location in the CONTROL is set to 0 and the control bit is advanced.

The run information at this point is:

```
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:

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\ansicpgl
```



```
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 0 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. Since the match length is less than 2, a literal is written to the output stream.

The control bit location in the CONTROL is set to 0 and the control bit is advanced.

The run information at this point is:

```
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:

WritePosition: 228

```

      0          1          2          3          4          5          6
0000  012345678901234567890123456789012345678901234567890123456789012345678901234
0065  {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0130  ss \fmodern \fscript \fdecor MS Sans SerifSymbolArialTimes New Ro
0195  manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
      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 0 to index 228 for the character ‘5’ results in 0 matches.

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

The control bit location in the CONTROL is set to 0 and the control bit is advanced.

Additionally, since the control bit is now 0x80 it is not advanced; rather, the run is now written to the output.

The run information at this point is:

```
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 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:

```
Output Cursor: 0x1b
```

```
000000: 00 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.

#### 4.2.1.3 Run 2

Prepare the next run by resetting the run information. The run information is now:

```
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=0229):

```
0          1          2          3          4          5          6
012345678901234567890123456789012345678901234567890123456789012345678901234
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:

```
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 dictionary reference (0x0af3) for match of length 5 at offset 175 (matching “\pard”); Run information is:

```
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 ' '; run information is:

```
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 literal 'h'; run information is:

```
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 literal 'e'; run information is:

```
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'; run information is:

```
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 match of length 2 at offset 144 (matching "lo"); run information is:

```
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 ' '. Since the control bit is 0x80, the run is now complete. Run information is:

```
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:

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

#### 4.2.1.4 Run 3

Prepare the next run by resetting the run information. The run information is now:

```
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=0242):

```
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
```

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

Add literal 'w'; run information is:

```
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 (0x0910) or match of length 2 at offset 91 (matching "or"); run information is:

```
Control Byte: 0x02
Control Bit: 0x04
Token Offset: 0x03
Token Buffer: 77 09 10 20 68 65 6c 09-00 20 00 00 00 00 00 00
```

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

Add literal 'l'; run information is:

```
Control Byte: 0x02
Control Bit: 0x08
Token Offset: 0x04
Token Buffer: 77 09 10 6c 68 65 6c 09-00 20 00 00 00 00 00 00
```

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

Add literal 'd'; run information is:

Control Byte: 0x02  
Control Bit: 0x10  
Token Offset: 0x05  
Token Buffer: 77 09 10 6c **64** 65 6c 09-00 20 00 00 00 00 00 00

Input:"}<CR><LF>"

Add literal '}' ; run information is:

Control Byte: 0x02  
Control Bit: 0x20  
Token Offset: 0x06  
Token Buffer: 77 09 10 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:

Control Byte: 0x22  
Control Bit: 0x40  
Token Offset: 0x08  
Token Buffer: 77 09 10 6c 64 7d **0a 80**-00 20 00 00 00 00 00 00

Input:<EMPTY>

Add dictionary reference for termination. Since the dictionary's write cursor is 250, the reference is 0x0fa0; Run information is:

Control Byte: 0x62  
Control Bit: 0x80  
Token Offset: 0x0a  
Token Buffer: 77 09 10 6c 64 7d 0a 80-**0f a0** 00 00 00 00 00 00 00

The run is now complete and is written to the output:

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
```

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

RAWSIZE: 43

COMPSIZE: 45

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

This results in the final output:

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
```

The output is 0x031 bytes long.

## 4.2.2 Example 2: Compressing with Tokens that Cross WritePosition

This example will compress the following RTF data.

```
“{\rtf1 WXYZWXYZWXYZWXYZWXYZ}”
```

### 4.2.2.1 Initialization

The dictionary is initialized with the data as described in section 3.3.3.1. After the initialization the dictionary is:

WritePosition: 207

```
      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
```

NonPrintable Characters:

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

CRC is: 0

COMPSIZE is: 0x000C

COMPTYPE is: 0x75465a4c

Output is:

Output Cursor: 0x10

```
000000: 00 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 00
000020: 00 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 00
```

InputCursor is: 0

### 4.2.2.2 Run 1

Start by initializing the 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 00
```

Input Data is “{\rtf1 WXYZWXYZWXYZWXYZWXYZ}”

Add dictionary reference (0x0004) for match of length 6 at offset 0 (matching “{\rtf1”); run information is:

```
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:

```
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:

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:

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. Thus, 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:

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 WXYZWXYZWXYZWXYZ}
```

```
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 "WXYZWXYZWXYZWXYZ"); run information is:

```
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 00
```

Input Data is now "}".

Add literal '>'; run information is:

```
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
```

Since the control bit was 0x80, the run is written to the output:

```
Output Cursor: 0x1b

000000: 00 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
```

### 4.2.2.3 Run 2

Start by initializing the run information:

```
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:

WritePosition: 235

```
      0      1      2      3      4      5      6
0000 01234567890123456789012345678901234567890123456789012345678901234
0065 {\rtf1\ansi\mac\deff0\deftab720{\fonttbl;}{\f0\fnil \froman \fswi
0130 manCourier{\colortbl\red0\green0\blue0__\par \pard\plain\f0\fs20\
0195 b|i\u\tab\tx{\rtf1 WXYZWXYZWXYZWXYZ}
```

```
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 0 is added for the WritePosition; the run is:

```
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:

```
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
000030: 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 0x002d)
```

This results in the final output:

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

The output is 0x1e bytes long.

### 4.3 Generating the CRC

#### 4.3.1 Example of CRC Generation

This example will compute the CRC of the following bytes (the compressed input from section 4.1.1, 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 use the procedure outlined in section 3.1.1.1.

##### 4.3.1.1 Initialization

The CRC is initially set to 0x00000000. The values in crcTableValue are also initialized (see section 3.1.3.2.1).

##### 4.3.1.2 First Byte

The first byte is 0x03, and the current CRC is 0x00000000, so tablePosition is computed as:

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

Using this to index into `crcTableValue`, getting a table value of:

```
tableValue         = 0x990951ba
```

Then compute the `intermediateValue` as:

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

The CRC that incorporates this initial byte is then:

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

### 4.3.1.3 Second Byte

The next byte is `0x00`, and the current CRC is `0x990951ba`, so `tablePosition` is computed as:

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

from which `tableValue` is:

```
tableValue         = 0x2bb45a92
```

The `intermediateValue` is then:

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

and the updated CRC is:

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

### 4.3.1.4 Continuation

The computation proceeds as above, incorporating each byte into the CRC. The final CRC of this set of input bytes is `0x98e32d45`.

## 5 Security

### 5.1 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 which 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.

### 5.2 Index of Security Parameters

None.

## 6 Appendix A: Office/Exchange Behavior

The information in this specification is applicable to the following versions of Office/Exchange:

- Office 2003 with Service Pack 3 applied
- Exchange 2003 with Service Pack 2 applied
- Office 2007 with Service Pack 1 applied
- Exchange 2007 with Service Pack 1 applied

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Office/Exchange behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies Office/Exchange does not follow the prescription.

---

<1> Section 3.2.4.1.1: Outlook 2003 SP3, Outlook 2007 SP1, Exchange 2003 SP2, and Exchange 2007 SP1 will read all bytes until the end of the stream is reached, regardless of the value of RAWSIZE.

<2> Section 3.3.4.2: When compressing zero (0) bytes of data, Outlook 2003 SP3, Outlook 2007 SP1, Exchange 2003 SP2, and Exchange 2007 SP1 will add a null in compression and the compressed run will be [02 00 0D 00] instead of [01 0C F0].

<3> Section 3.3.4.2.1: Multiple mechanisms can be used for locating the longest match in a buffer. Outlook 2003 SP3, Outlook 2007 SP1, Exchange 2003 SP2, and Exchange 2007 SP1 utilize an alternate mechanism which complies with the requirements specified in this protocol.

## 7 Index

- Applicability statement, 6
- Common details, 10
- Compression details, 15
- Decompression details, 13
- Examples, 20
- Fields, vendor-extensible, 7
- Glossary, 5
- Index of security parameters, 42
- Informative references, 6
- Introduction, 5
- Message syntax, 7
- Messages, 7
  - Message syntax, 7
  - Transport, 7
- Normative references, 6
- Office/Exchange behavior, 43
- Overview, 6
- Preconditions, 6
- Prerequisites, 6
- Protocol details, 10
  - Common details, 10
  - Compression details, 15
  - Decompression details, 13
- References, 6
  - Informative references, 6
  - Normative references, 6
- Relationship to other protocols, 6
- Security, 42
  - Considerations for implementers, 42
  - Index of security parameters, 42
- Security considerations for implementers, 42
- Standards assignments, 7
- Transport, 7
- Vendor-extensible fields, 7
- Versioning and capability negotiation, 7