Network Working Group
INTERNET-DRAFT
Expires: May 2002

Richard Price, Siemens/Roke Manor
Jonathan Rosenberg, dynamicsoft
Abigail Surtees, Siemens/Roke Manor
Mark A West, Siemens/Roke Manor
Lawrence Conroy, Siemens/Roke Manor

14 November, 2001

Universal Decompression Algorithm
<draft-ietf-rohc-sigcomp-algorithm-00.txt>

Status of this Memo

This document is a submission to the IETF ROHC WG. Comments should be directed to the mailing list of ROHC, rohc@cdt.luth.se.

Abstract

This specification defines a "universal decompressor" capable of interoperating with a wide range of compression algorithms. Using the basic techniques of Huffman compression and LZ77-style string substitution, the decompressor can be configured to understand the output of many well-known compressors including [DEFLATE] and [LZW].

Revision history

Changes from <draft-ietf-rohc-sigcomp-01.txt>

New COPY-LITERAL and COPY-OFFSET tokens added to reduce complexity for LZ77-based algorithms.

COMPARE token modified to allow it to be used without a SWITCH token.

HUFFMAN token modified to require fewer parameters.

Support added for literal parameters (see Section 4.3).

Example token sets added for decompression of LZ77, [DEFLATE] and [LZW] compressed messages.

Table of contents

1. Introduction

This draft introduces the concept of a "universal decompressor".

The goal of the document is to standardize a decompressor capable of interoperating with a wide range of compression algorithms. Consequently this draft describes the decompressor operation only, i.e. the actions which the decompressor takes upon receiving a certain instruction from the compressor.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC-2119].

Byte buffer

The universal decompressor maintains a byte buffer containing any previously received text strings that might be useful for future compression.

Token

A token is an instruction transmitted from the compressor to the decompressor.

3. Requirements on underlying transport protocol

The universal decompressor takes as its input a sequence of compressed messages, which are processed and then outputted as a sequence of uncompressed messages. This chapter lists the requirements on the transport protocol used to carry compressed messages to the universal decompressor.

Note that the universal decompressor outputs an uncompressed message when it encounters an explicit end-of-message character. Consequently there is no need for one uncompressed message to correspond to exactly one compressed message. Two or more compressed messages can be sent to reconstruct a single uncompressed message, which is very useful for segmenting compressed messages that are larger than the MTU of the transport protocol.

Conversely, one compressed message can be sent to reconstruct several uncompressed messages. In particular, messages can be successfully decompressed even when the transport protocol provides data as a byte stream with no framing.

Note however that the universal decompressor can make use of

previously received messages to improve the overall compression ratio (see [V-J] for an example of a compression algorithm which uses previously received messages in this manner).

Consequently, the main requirement on the underlying protocol is that it MUST ensure that the contents of the decompressor byte buffer are as expected by the next message to be decompressed.

This requirement can be supported in a number of ways. If the underlying transport protocol is reliable (for example TCP or [SCTP]) then the compressed messages are provided to the universal decompressor in the correct order and free from bit errors. In this case the byte buffer is automatically updated correctly between messages.

If the underlying transport protocol is unreliable (for example UDP) then the byte buffer MUST be reset to a known state between compressed messages. This ensures that messages are not compressed relative to text that has not yet been received by the universal decompressor.

The "known state" of the byte buffer can be negotiated a-priori: for example a set of tokens can be downloaded from the compressor to the decompressor when the universal decompressor is being set up. All messages are then compressed relative to these tokens. Alternatively the transport protocol can indicate to the decompressor how the byte buffer should be set up before each message is decompressed.

4. Description of the universal decompressor

An important feature of the universal decompressor is that it can interoperate with a wide range of compression algorithms. The precise method for compression is left as an implementation decision, and in fact the standard decompressor can interoperate with any of the following classes of algorithm:

* Generic text compressor (for example [DEFLATE] or a similar algorithm).

* Protocol-aware compressor offering excellent performance for one type of text message (for example the text messages generated by [SIP]).

* Hybrid compressor with similar performance to [DEFLATE] for generic text strings and superior performance for certain types of text message.

The choice of which tokens to send to the decompressor is left as a local implementation decision at the compressor. The only requirement is that of transparency, i.e. the compressor MUST NOT send tokens which cause the decompressor to incorrectly decompress a given message.

Note however that it is perfectly acceptable for the compressor to

send tokens which update the byte buffer at the decompressor, but which cause no decompressed message to be outputted. Indeed, this is a useful technique for pre-populating the dictionary with well-known text strings.

4.1. Structure of universal decompressor dictionary

The universal decompressor dictionary consists of a simple byte buffer designed to hold the current uncompressed message, the current compressed message, and any other previously received text strings that might be useful for future compression.

The size buffer_size of the byte buffer can be negotiated by an externally defined mechanism (e.g. by the underlying protocol used to transport the compressed messages). Entries in the byte buffer are referred to as buffer[n] where 0 =< n < buffer_size.

As all of the tokens currently use 2-byte indices into the byte buffer, the maximum size of the buffer is 64K.

4.2. Important entries in the byte buffer

The first few bytes in the byte buffer are used to store some important 2-byte integers. These integers are given the following names:

Position in buffer:	Name:
0 - 1	first_token
2 - 3	uncompressed_start
4 - 5	uncompressed_end
6 - 7	circular_buffer
8 - 9	compressed_start
10 - 11	compressed_end
12 - 13	compressed_pointer
14 - 15	stack_free
16 - 17	stack[0]
18 - 19	stack[1]
20 - 21	stack[2]
:	:

The MSBs of the integer are always stored before the LSBs. So, for example, the MSBs of first_token are stored in buffer[0] whilst the LSBs are stored in buffer[1].

The use of each integer is described in the following sections of the draft.

4.3. Token parameters

Each of the tokens is followed by 0 or more bytes containing the parameters required by the token. At present all parameters are stored as 2-byte integers with MSBs stored in the byte preceding the LSBs in the byte buffer.

The most significant bit of the 2-byte integer has a special meaning: it is used to determine whether the parameter is a literal value or an index pointing to a literal value.

If the most significant bit is 0 then the 2-byte integer is interpreted as a literal value from 0 to 32767.

If the most significant bit is 1 then the 2-byte integer is interpreted as an index from 0 to 32767 (the most significant bit itself is ignored). This index points to the location in the buffer containing the literal value of the parameter (MSBs stored in buffer[index] and LSBs stored in buffer[index + 1]). If the index references a location beyond the size of the byte buffer then a bad compressed message has been received and decompression failure occurs (see Section 4.5.).

4.4. Decompressor actions upon receiving a compressed message

When the universal decompressor is initialized all entries in the byte buffer are set to 0. Upon receiving a compressed message, the decompressor strips off the underlying protocol header and then performs the following actions:

1.) The message is copied directly into the byte buffer beginning at the byte specified in compressed_start.

The underlying protocol MUST NOT pass a compressed message of more than 1460 bytes to the universal decompressor. If a larger compressed message is received, the underlying protocol passes only the first 1460 bytes to the decompressor, and provides additional working memory to store the bytes that are not currently being decompressed. The remainder of the compressed message is passed to the decompressor in blocks of 1460 bytes (or less if it is the last block in the compressed message).

The decompressor MUST NOT concatenate two messages to form a single compressed message. This is because compressed messages are typically padded with trailing zero bits so that they are a whole number of bytes long. Concatenating two messages would cause these padding bits to be incorrectly interpreted as compressed data.

Note that the buffer is circular, so once a byte is copied into buffer[buffer_size - 1], the next byte is copied into buffer[circular_buffer]. The parameter circular_buffer (see Section 4.2) can be set to prevent the first part of the buffer from being overwritten by new messages. Typically this area of the buffer is used to hold important tokens and text strings that should be kept from one compressed message to the next. If circular_buffer lies beyond the size of the byte buffer then decompression failure occurs (see Section 4.5).

After the message has been copied into the byte buffer, the position of the last byte in the compressed message is copied into

compressed_end. Also, the value in compressed_start is copied into compressed_pointer.

2.) Next, the tokens contained within the byte buffer are executed beginning at the byte specified in first_token. The tokens are

executed consecutively unless indicated explicitly (for example when the decompressor encounters a SWITCH token). If the next token to be executed lies outside the byte buffer then decompression failure occurs (see Section 4.5).

3.) The decompressor stops token execution when it reaches buffer[0] or buffer[1]. Depending on which buffer entry is reached, the following actions are then taken:

If the decompressor reaches buffer[0], instead of executing the token contained within buffer[0] it stops token execution and outputs the uncompressed message. The location of the uncompressed message is from uncompressed_start up to, but not including uncompressed_end. After the uncompressed message has been outputted, the value in uncompressed_end is then copied into uncompressed_start. If uncompressed_start = uncompressed_end then no uncompressed message is outputted.

As stated before the buffer is circular, so once a byte is copied from buffer[buffer_size - 1], the next byte is copied from buffer[circular_buffer]. If either uncompressed_start or uncompressed_end lie outside the circular buffer then decompression failure occurs.

If buffer[1] is reached then token execution stops but no uncompressed message is outputted.

When the next compressed message becomes available, the universal decompressor continues at Step 1.) above. Note that if the underlying transport protocol does not provide reliable, in-order message delivery then the contents of the byte buffer MUST be reset to the state expected by the next compressed message. This state can be negotiated a-priori, or the transport protocol can indicate to the decompressor how the byte buffer should be set up before the next message is decompressed.

4.5. Decompression failure

If the compressed messages received by the decompressor are corrupted (either accidentally or maliciously) then one of three possibilities might occur:

* A decompressed message is outputted that is incorrect.

* A token is encountered that cannot be processed successfully by the decompressor (for example a RETURN token when no CALL token has previously been encountered).

* The decompressor never finishes decompressing a message.

To counter the first possibility the underlying protocol SHOULD include a checksum to verify either the compressed message or the uncompressed message. If a message fails the checksum then "decompression failure" has occurred. The decompressor does not

output an uncompressed message, and ignores any future compressed message until the byte buffer is reset.

If a token is encountered that cannot be successfully processed then decompression failure occurs automatically.

To counter the third possibility, decompression failure SHOULD also occur after a certain number of tokens have been processed for a given compressed message. The maximum number of tokens to process is currently left as an implementation decision (but might in future be negotiated).

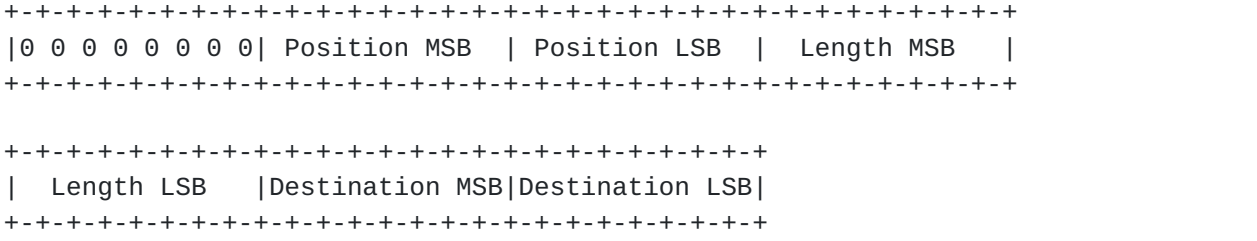
5. Library of tokens

The universal decompressor currently understands twelve types of token, chosen to support the widest possible range of compression algorithms with the minimum possible overhead.

All tokens are stored as a single byte to indicate the token type, followed by 0 or more bytes containing the parameters required by the token. At present all parameters are 2-byte integers with MSBs stored before LSBs. For example, the COPY token is followed by three parameters as shown below:

COPY (position, length, destination)

In the byte buffer a COPY token is stored as the following 7 bytes:



Twelve token types are currently available to the universal decompressor. The following table lists the different token types and the byte values used to transmit the tokens to the decompressor:

Token type:	Corresponding byte value:
COPY	0
COPY-LITERAL	1
COPY-OFFSET	2
ADD	3
SUBTRACT	4
LSHIFT	5
RSHIFT	6
COMPARE	7

SWITCH	8
CALL	9
RETURN	10
HUFFMAN	11

Each token is explained in more detail below:

5.1. COPY

The COPY token instructs the decompressor to copy a string of bytes from one part of the byte buffer to another.

A COPY token is stored in the byte buffer as 7 consecutive bytes as follows:

COPY (position, length, destination)

As with all tokens currently defined, the COPY token translates into a single byte for the token type followed by 2 bytes for each of the parameters.

The meaning of the three parameters is explained below:

Position: 2-byte integer indicating the location of the first byte in the string to be copied.

Length: 2-byte integer indicating the number of bytes to be copied.

Destination: 2-byte integer indicating the location to which the first byte in the string will be copied.

Note that the copying function is performed on a byte-by-byte basis, with the position parameter indicating the first byte to be copied. In particular, some of the later bytes to be copied may themselves have been written into the byte buffer by the COPY token currently being executed.

Equally, it is possible for a COPY token to overwrite itself or its parameters. If this occurs then the COPY token MUST continue to execute as if the parameters were still in place in the byte buffer.

If the source or destination of the next byte to be copied is larger than (buffer_size - 1) then sufficient multiples of (buffer_size - circular_buffer) are subtracted until it is not. So for example, once a byte is copied into buffer[buffer_size - 1] the next byte is copied into buffer[circular_buffer]. If circular_buffer equals or exceeds buffer_size then a bad compressed message has been received and decompression failure occurs (see Section 4.5).

A modified version of the COPY token is given below:

COPY-LITERAL (position, length, destination)

The COPY-LITERAL token is identical to COPY except that after

copying, the destination parameter is replaced with the value (destination + length). If the destination parameter is a literal value then it is updated directly. If the destination parameter is an index then the literal value it references is updated instead.

As above, if (destination + length) is larger than (buffer_size - 1) then sufficient multiples of (buffer_size - circular_buffer) are subtracted until it is not.

A further version of the COPY-LITERAL token is given below:

COPY-OFFSET (offset, length, destination)

The COPY-OFFSET token is identical to COPY-LITERAL except that an offset parameter is given instead of a position parameter. The two parameters are related by position = (destination - offset).

If (destination - offset) does not lie between circular_buffer and (buffer_size - 1) inclusive then sufficient multiples of (buffer_size - circular_buffer) are added or subtracted until it does.

5.2. ADD / SUBTRACT

The ADD token instructs the decompressor to add two 2-byte integers (addition performed modulo 2^16) and to store the result in the location of the first parameter.

The format of the ADD token is given below:

ADD (parameter_1, parameter_2)

Note that as per Section 4.3, depending on how the MSB is set the parameters can be interpreted as literal values or indices to literal values. If parameter_1 is a literal value then it is overwritten with the result of the addition. If parameter_1 is an index to a literal value then the literal value is overwritten (not parameter_1 itself).

The SUBTRACT token is the same as the ADD token except that the second integer is subtracted from the first (subtraction performed modulo 2^16):

SUBTRACT (parameter_1, parameter_2)

5.3. LSHIFT / RSHIFT

The LSHIFT token instructs the decompressor to left shift a 2-byte value by the specified number of bits:

LSHIFT (parameter, no_of_bits)

The value to be shifted is stored in parameter, and the number of bits to shift is stored in no_of_bits. As usual both can be stored either as a literal value or as an index to a literal value.

If the value of no_of_bits does not lie between 0 and 15 inclusive then a bad compressed message has been received and decompression failure occurs.

The RSHIFT token is the same as the LSHIFT token except that the bits are right shifted:

RSHIFT (parameter, no_of_bits)

5.4. COMPARE

The COMPARE token instructs the decompressor to compare two 2-byte values and then to jump to one of three specified indices depending on the result.

COMPARE (parameter_1, parameter_2, index_1, index_2, index_3)

If the parameter_1 < parameter_2 then the decompressor continues token execution at the byte position specified by index 1. If parameter_1 = parameter_2 then it jumps to index_2. If parameter_1 > parameter_2 then it jumps to index_3.

If an index is specified which is beyond the size of the byte buffer, a bad compressed message has been received and decompression failure occurs.

5.5. SWITCH

The SWITCH token performs a conditional jump based on the value of its first parameter.

SWITCH (j, index_0, index_1, ... , index_n-1)

When a SWITCH token is encountered the decompressor reads the value of j. It then continues token execution at the byte position specified by index j.

If j specifies an index which is beyond the size of the byte buffer, a bad compressed message has been received and decompression failure occurs.

5.6. CALL / RETURN

The CALL and RETURN tokens provide support for compression algorithms with a nested structure.

CALL (index)

RETURN

When the decompressor reaches a CALL token, it finds the byte position of the token immediately following the CALL token and copies this 2-byte integer into stack[stack_free] ready for later retrieval. It then increases stack_free by 1 and continues token

execution at the byte position specified by index.

When the decompressor reaches a RETURN token it decreases stack_free by 1, and then continues token execution at the byte position stored in stack[stack_free].

If stack_free ever becomes more than buffer_size - 1 or less than 0 then a bad compressed message has been received and decompression failure occurs (see Section 4.5).

5.7. HUFFMAN

The HUFFMAN token maps a shorthand Huffman code onto its uncompressed equivalent.

The format of a HUFFMAN token is as follows:

HUFFMAN (position, bit_offset, destination, n, bits_1, uncomp_1, code_1, bits_2, uncomp_2, code_2, ... , bits_n-1, uncomp_n-1, code_n-1, bits_n, uncomp_n)

The HUFFMAN token is followed by four mandatory parameters plus n additional sets of parameters. Every set contains three parameters except the nth set, which contains two. The nth set of parameters omits the parameter code_n, as the decompressor can always work out what the correct value of code_n should be.

The meaning of the four mandatory parameters is explained below:

Position: Indicates the byte location of the Huffman code to be decompressed.

Bit Offset: Indicates the bit offset at which the Huffman code begins within the byte specified above.

Destination: Indicates the location to which the uncompressed value will be copied.

n: Indicates the number of additional sets of parameters that follow. If n = 0 then the HUFFMAN token is ignored by the decompressor.

The remaining parameters specify the actual Huffman codes and their uncompressed equivalents. Each set of 3 parameters specifies a block of Huffman codes with the same length and (when treated as integers) with values that increase by 1 for each additional Huffman code. The precise meaning of the parameters is given below:

bits_j: Indicates the additional length in bits of the Huffman codes in block j, compared to the Huffman codes in block j-1. Note that the total length of the Huffman codes in set j is (bits_1 + bits_2 + ... + bits_j).

uncomp_j: Indicates the uncompressed value of the first Huffman code in set j.

code_j: Indicates the compressed value (when read as an integer) of the last Huffman code in set j.

Note that the compressed value of the first Huffman code in set j is not specified explicitly, but instead is taken to be 0 when j = 1 or (2^bits_j) x (1 + code_j-1) when j > 1. This simply means that the Huffman code is 1 greater than the largest Huffman code in set j-1, but padded with enough zeroes to give it the correct length in bits.

For example, suppose that bits_1 = 2, uncomp_1 = 15 and code_1 = 1. This defines a set of 2 Huffman codes (00 and 01) with corresponding values 15 and 16.

Suppose also that bits_2 = 1, uncomp_2 = 4 and code_2 = 7. This defines an additional set of 4 Huffman codes (100, 101, 110 and 111) with corresponding values 4, 5, 6 and 7.

Huffman code:	Uncompressed value:
00	15
01	16
100	4
101	5
110	6
111	7

The motivation for downloading Huffman codes to the decompressor in this form is that it is very easy to convert a compressed Huffman code into its uncompressed equivalent. This can be achieved by taking the following steps:

1.) Set j = 1, set code_0 = 65535 and set code_n = 65535.

2.) Read a total of (bits_1 + bits_2 + ... + bits_j) bits starting from the specified position and bit_offset. Interpret these bits as an integer H, with the first bit to be read as the MSB and the last bit to be read as the LSB.

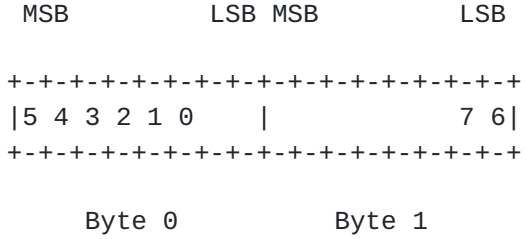
3.) If (H > code_j) then set j = j + 1 and goto 2.

4.) Output (H + uncomp_j - (2^bits_j) x (1 + code_j-1)), with the arithmetic operations calculated modulo 2^16.

Note that as the HUFFMAN token reads individual bits from within a byte, to avoid ambiguity it is necessary to define the order in which these bits are read. This draft specifies that a bit offset of 0 indicates the least significant bit of a byte, whilst a bit offset of 7 indicates the most significant bit.

For example, suppose that an 8-bit Huffman code begins at byte

position 0 and bit offset 2. In this case the 8 bits of the Huffman code can be found in the following locations (Bit 0 is the first bit in the Huffman code and Bit 7 is the last bit):



If the parameter bit offset does not take a value between 0 and 7 inclusive then a bad compressed message has been received and decompression failure occurs (see Section 4.5.). Decompression failure also occurs if a total of more than 16 bits are read from the specified position and bit offset.

6. Security considerations

Note that this initial version raises the issues, without detailing a specific solution to resolve them.

Introduction:

In effect, a compressed message is a program; the tokens are instructions that are executed by the decompressor. This presents particular risks to the operation of the node on which the decompressor is running. This is the case for any compression algorithm, and so affects the operation of a node using this one.

Transport Mechanism Requirements:

The algorithm itself has no security issues, but does place requirements on any transport mechanism used to deliver the messages.

If such requirements are not met, then the operation of the system can be exposed to attack ranging from indirect reduction in service, direct denial of service, and modification of subsequent messages.

An initial list of requirements on the transport mechanism is:

- messages should be delivered reliably to avoid corruption - for some uses of this algorithm, this corruption may have longer lasting effects.

- messages should be not be duplicated - to ensure that operations implied by those messages are executed once only.

- potentially, the message source should be identified and/or validated - to restrict possible insinuation of "attack" messages by third parties and to allow "blacklisting" of individual sources that

"behave badly".

7. References

[DEFLATE] "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, P. Deutsch, May 1996

[V-J] "Compressing TCP/IP Headers for Low-Speed Serial Links", V. Jacobson, Internet Engineering Task Force, February 1990

[LZW] "LZW Data Compression", Mark Nelson, Dr. Dobb's Journal, October 1989

[SCTP] "Stream Control Transmission Protocol", Stewart et al, RFC 2960, Internet Engineering Task Force, October 2000

[SIP] "SIP: Session Initiation Protocol", Handley et al, RFC 2543, Internet Engineering Task Force, March 1999

8. Authors' addresses

Richard Price Tel: +44 1794 833681
Email: richard.price@roke.co.uk

Abigail Surtees Tel: +44 1794 833131
Email: abigail.surtees@roke.co.uk

Mark A West Tel: +44 1794 833311
Email: mark.a.west@roke.co.uk

Lawrence Conroy Tel: +44 1794 833666
Email: lwc@roke.co.uk

Roke Manor Research Ltd
Romsey, Hants, SO51 0ZN
United Kingdom

Jonathan Rosenberg
dynamicsoft
72 Eagle Rock Avenue
First Floor
East Hanover, NJ 07936
Email: jdrosen@dynamicsoft.com

Appendix A. Example sets of tokens

This appendix gives three example sets of tokens which can be downloaded to the universal decompressor. The first set of tokens can be used to decompress a very simple variant of LZ77, which illustrates how the different token types are intended to be used.

The next two examples show how the universal decompressor can be configured to understand the output of existing compression algorthms. One set of tokens allows the decompressor to understand [DEFLATE] compressed messages, whilst the other set allows the decompressor to understand [LZW] compressed messages.

A.1. Mnemonic language

Presenting the example set of tokens as a set of bytes would not be very informative, so they are written instead using a simple mnemonic language. Most importantly, the language allows the parameters of a token to be specified as text references rather than as 2-byte integers.

The mnemonic language is translated into bytes as follows:

Tokens: Token names are given in capitals. Replace each name with the corresponding 1-byte value as per Chapter 5.

Labels: Label names are given as a colon followed by lowercase text. They are deleted when converting the mnemonics to bytes.

References: These are indicated by a label name without the colon. Replace each reference with a 2-byte value specifying the location of the byte immediately following the label. Moreover if the reference is preceded by a "$" symbol then it is interpreted as an pointer rather than a literal value, and hence the MSB of the 2-byte value is set to 1.

.short: Each decimal number following ".short" is interpreted as a 2-byte integer with MSBs preceding LSBs. Unless otherwise specified, this is the default.

.byte: Each decimal number following ".byte" is interpreted as a single byte.

Whitespace: All whitespace, brackets and commas just delimit the tokens. Delete.

Comments: These are indicated by a semicolon and continue

to the end of the line. Delete.

A.2. Example token set for simple LZ77 decompression

The first example gives the tokens required to support a very simple LZ77-based algorithm. The decompressor is instructed to interpret a compressed message as a set of 4-byte characters, where each character contains a 2-byte position integer followed by a 2-byte length integer. Taken together these integers point to a previously received text string in the byte buffer, which is then copied to the end of the uncompressed message.

Since the compressor can only send references to strings already present in the byte buffer, before the first message is decompressed the buffer must be initialized with a static dictionary containing the 256 ASCII characters.

```
:first_token            unpack_static_dictionary
:uncompressed_start     1792
:uncompressed_end       2048
:circular_buffer        2048
:compressed_start       compressed_message_start
:compressed_end         0
:compressed_pointer     0

:bit_offset             0
:position               0
:length                 0
:token_start            next_character

:unpack_static_dictionary

COPY-LITERAL (17, 1, $uncompressed_start)
ADD ($position, 1)
COMPARE ($position, 256, unpack_static_dictionary, continue, 0)

; The above tokens are used to initialize the static dictionary.

:continue

COPY (token_start, 2, first_token)

:next_character

HUFFMAN ($compressed_pointer, $bit_offset, position, 1, 16, 0)
HUFFMAN ($compressed_pointer, $bit_offset, length, 1, 16, 0)
COPY-LITERAL ($position, $length, $uncompressed_end)
COMPARE ($compressed_pointer, $compressed_end, next_character, 0, 0)

:compressed_message_start
```

A.3. Example token set for DEFLATE decompression

This example gives the tokens required to understand [DEFLATE] compressed messages as defined in RFC 1951. Note that the example

only covers static Huffman codes (Block Type 01); dynamic Huffman codes will be added in a future version.

Note also that the order in which the HUFFMAN token decompresses bits within a byte differs from that specified in RFC 1951. An alternative version of the HUFFMAN token may be introduced to support the exact bit order of [DEFLATE] in future (although this is not an urgent issue, as the order in which bits are decompressed does not affect the overall compression ratio).

```
:first_token            next_character
:uncompressed_start     2048
:uncompressed_end       2048
:circular_buffer        2048
:compressed_start       compressed_message_start
:compressed_end         0
:compressed_pointer     0

:bit_offset             0
:index                  0
:extra_length_bits      0
:length_value           0
:extra_distance_bits    0
:distance_value         0

:next_character

HUFFMAN ($compressed_pointer, $bit_offset, index, 4, 7, 16428, 23,
         1, 0, 191, 0, 16452, 199, 1, 144)

; The above HUFFMAN token decompresses a Huffman code from the
; length/literal alphabet.

COMPARE ($index, 16428, literal, 0, length)

; The COMPARE token is used to determine whether the decoded
; character should be interpreted as a length value, a literal value
; or an end-of-block character. In the latter case the decompressor
; stops and outputs the uncompressed message.

:literal

COPY-LITERAL (17, 1, $uncompressed_end)
COMPARE ($compressed_pointer, $compressed_end, next_character, 1, 1)

; If the decoded character is to be interpreted as a literal value
; then it is copied directly to the uncompressed message. The
; decompressor then checks whether more compressed characters are
; available. If it has run out then it pauses and waits for some
; more.
```

```
   :length

   LSHIFT ($index, 2)
```

```
   COPY ($index, 4, extra_length_bits)

   ; If the decoded character is to be interpreted as a length value
   ; then as specified in [DEFLATE], the decompressor must first add a
   ; certain number of additional bits from the compressed message to
   ; this length value.

   HUFFMAN ($compressed_pointer, $bit_offset, extra_length_bits, 1,
            $extra_length_bits, 0)

   ; The above HUFFMAN token obtains the correct number of additional
   ; length bits from the compressed message.

   ADD ($length_value, $extra_length_bits)

   :distance

   HUFFMAN ($compressed_pointer, $bit_offset, index, 1, 5, 74)

   ; In [DEFLATE] a length code is always followed by a distance code.
   ; The above HUFFMAN token decompresses this character.

   LSHIFT ($index, 2)
   COPY ($index, 4, extra_distance_bits)
   HUFFMAN ($compressed_pointer, $bit_offset, extra_distance_bits, 1,
            $extra_distance_bits, 0)

   ; The above HUFFMAN token obtains the correct number of additional
   ; distance bits from the compressed message.

   ADD ($distance_value, $extra_distance_bits)
   COPY-OFFSET ($distance_value, $length_value, $uncompressed_end)

   ; The text string specified by the length and distance codes is then
   ; copied to the end of the uncompressed message.

   COMPARE ($compressed_pointer, $compressed_end, next_character, 1, 1)

   .byte 0 0 0 .short    ; 3 bytes worth of padding

   :length_table

   0       3       0       4       0       5
   0       6       0       7       0       8
   0       9       0       10      1       11
   1       13      1       15      1       17
   2       19      2       23      2       27
   2       31      3       35      3       43
   3       51      3       59      4       67
   4       83      4       99      4       115
```

```
   5       131     5       163     5       195
   5       227     0       258

   ; This is the length table from Page 11 of [DEFLATE].
```

```
   :distance_table

   0       1       0       2       0       3
   0       4       1       5       1       7
   2       9       2       13      3       17
   3       25      4       33      4       49
   5       65      5       97      6       129
   6       193     7       257     7       385
   8       513     8       767     9       1025
   9       1537    10      2049    10      3073
   11      4097    11      6145    12      8193
   12      12289   13      16385   13      24577

   ; This is the distance table from Page 11 of [DEFLATE].

   :compressed_message_start

   ; The compressed messages are stored in the byte buffer following
   ; the above set of tokens.
```

A.4. Example token set for LZW decompression

This example gives the tokens required to understand [LZW] compressed messages. This particular variant of LZW uses a codebook containing up to 1024 entries, and so each compressed character is a 10-bit value referencing one of the entries in the codebook.

```
   :first_token            unpack_static_dictionary
   :uncompressed_start     5888
   :uncompressed_end       1
   :circular_buffer        6144
   :compressed_start       compressed_message_start
   :compressed_end         0
   :compressed_pointer     0

   :bit_offset             0
   :index                  0
   :position_value         0
   :length_value           0
   :codebook_next          2048
   :token_start            next_character

   :unpack_static_dictionary

   COPY-LITERAL (uncompressed_start, 4, $codebook_next)
   COPY-LITERAL (17, 1, $uncompressed_start)
   ADD ($index, 1)
   COMPARE ($index, 256, unpack_static_dictionary, continue, 0)

   ; The above tokens are used to initialize the first 256 entries in
```

```
   ; the LZW codebook with single ASCII characters.

   :continue
```

```
   COPY (circular_buffer, 2, uncompressed_end)
   COPY (token_start, 2, first_token)

   :next_character

   HUFFMAN ($compressed_pointer, $bit_offset, index, 1, 10, 512)

   ; The above HUFFMAN token extracts 10 bits from the compressed
   ; message.

   LSHIFT ($index, 2)
   COPY ($index, 4, position_value)

   ; The contents of the corresponding codebook entry is retrieved by
   ; the above COPY token.

   COPY-LITERAL (uncompressed_end, 2, $codebook_next)
   COPY-LITERAL ($position_value, $length_value, $uncompressed_end)

   ; The above COPY-LITERAL token appends the selected text string to
   ; the end of the uncompressed message.

   ADD ($length_value, 1)
   COPY-LITERAL (length_value, 2, $codebook_next)

   ; As per the LZW algorithm, a new entry is added to the codebook
   ; containing the text string copied to the end of the uncompressed
   ; message, plus the next character in the uncompressed message.

   COMPARE ($compressed_pointer, $compressed_end, next_character, 0, 0)

   :compressed_message_start
```