Network Working Group
Request for Comments: 4464
Category: Informational

A. Surtees
M. West
Siemens/Roke Manor Research

May 2006

Signaling Compression (SigComp) Users' Guide

Status of This Memo

Copyright Notice

Abstract

This document provides an informational guide for users of the Signaling Compression (SigComp) protocol. The aim of the document is to assist users when making SigComp implementation decisions, for example, the choice of compression algorithm and the level of robustness against lost or misordered packets.

Table of Contents

1. Introduction

This document provides an informational guide for users of the SigComp protocol, RFC 3320 [2]. The idea behind SigComp is to standardize a Universal Decompressor Virtual Machine (UDVM) that can be programmed to understand the output of many well-known compressors including DEFLATE [8] and LZW [7]. The bytecode for the chosen compression algorithm is uploaded to the UDVM as part of the compressed data.

The basic SigComp RFC describes the actions that an endpoint must take upon receiving a SigComp message. However, the entity responsible for generating new SigComp messages (the SigComp compressor) is left as an implementation decision; any compressor can be used provided that it generates SigComp messages that can be successfully decompressed by the receiving endpoint.

This document gives examples of a number of different compressors that can be used by the SigComp protocol. It also gives examples of how to use some of the mechanisms (such as acknowledgements) described in RFC 3321 [3].

2. Overview of the User Guide

When implementing a SigComp compressor, the first step is to choose a compression algorithm that can encode the application messages into a (hopefully) smaller form. Since SigComp can upload bytecode for new algorithms to the receiving endpoint, arbitrary compression algorithms can be supported provided that suitable bytecode has been written for the corresponding decompressor.

This document provides example bytecode for the following algorithms:

1. LZ77
2. LZSS
3. LZW
4. DEFLATE
5. LZJH

Any of the above algorithms may be useful depending on the desired compression ratio, processing and memory requirements, code size, implementation complexity, and Intellectual Property (IPR) considerations.

As well as encoding the application messages using the chosen algorithm, the SigComp compressor is responsible for ensuring that messages can be correctly decompressed even if packets are lost or misordered during transmission. The SigComp feedback mechanism can

be used to acknowledge successful decompression at the remote endpoint.

The following robustness techniques and other mechanisms specific to the SigComp environment are covered in this document:

1. Acknowledgements using the SigComp feedback mechanism
2. Static dictionary
3. Cyclic redundancy code (CRC) checksum
4. Announcing additional resources
5. Shared compression

Any or all of the above mechanisms can be implemented in conjunction with the chosen compression algorithm. An example subroutine of UDVM bytecode is provided for each of the mechanisms; these subroutines can be added to the bytecode for one of the basic compression algorithms. (Note: The subroutine or the basic algorithm may require minor modification to ensure they work together correctly.)

3. UDVM Assembly Language

Writing UDVM programs directly in bytecode would be a daunting task, so a simple assembly language is provided to facilitate the creation of new decompression algorithms. The assembly language includes mnemonic codes for each of the UDVM instructions, as well as simple directives for evaluating integer expressions, padding the bytecode, and so forth.

The syntax of the UDVM assembly language uses the customary two-level description technique, partitioning the grammar into a lexical and a syntactic level.

3.1. Lexical Level

On a lexical level, a string of assembly consists of zero or more tokens optionally separated by whitespace. Each token can be a text name, an instruction opcode, a delimiter, or an integer (specified as decimal, binary, or hex).

The following ABNF description, RFC 4234 [1], specifies the syntax of a token:

```
token           =     (name / opcode / delimiter / dec / bin / hex)
name            =     (lowercase / "_") 0*(lowercase / digit / "_")
opcode          =     uppercase *(uppercase / digit / "-")
delimiter       =     "." / "," / "!" / "$" / ":" / "(" / ")" /
                      operator
dec             =     1*(digit)
bin             =     "0b" 1*("0" / "1")
hex             =     "0x" 1*(hex-digit)
hex-digit       =     digit / %x41-46 / %x61-66
digit           =     %x30-39
uppercase       =     %x41-5a
lowercase       =     %x61-7a
operator        =     "+" / "-" / "*" / "/" / "%" / "&" / "|" /
                      "^" / "~" / "<<" / ">>"
```

When parsing for tokens, the longest match is applied, i.e., a token is the longest string that matches the <token> rule specified above.

The syntax of whitespace and comments is specified by the following ABNF:

```
ws              =     *(%x09 / %x0a / %x0d / %x20 / comment)
comment         =     ";" *(%x00-09 / %x0b-0c / %x0e-ff)
                      (%x0a / %x0d)
```

Whitespace that matches <ws> is skipped between tokens, but serves to terminate the longest match for a token.

Comments are specified by the symbol ";" and are terminated by the end of the line, for example:

```
LOAD (temp, 1) ; This is a comment.
```

Any other input is a syntax error.

When parsing on the lexical level, the string of assembly should be divided up into a list of successive tokens. The whitespace and comments should also be deleted. The assembly should then be parsed on the syntactic level as explained in Section 3.2.

3.2. Syntactic Level

Once the string of assembly has been divided into tokens as per Section 3.1, the next step is to convert the assembly into a string of UDVM bytecode.

On a syntactic level, a string of assembly consists of zero or more instructions, directives, or labels, each of which is itself built up from one or more lexical tokens.

The following ABNF description specifies the syntax of the assembly language. Note that the lexical parsing step is assumed to have been carried out; so in particular, the boundaries between tokens are already known, and the comments and whitespace have been deleted:

```
assembly          =    *(instruction / directive / label)
instruction       =    opcode ["(" operand *("," operand) ")"]
operand           =    [["$"] expression]
                           ; Operands can be left blank if they can
                           ; be automatically inferred by the
                           ; compiler, e.g., a literal operand
                           ; that specifies the total number of
                           ; operands for the instruction.
                           ; When "$" is prepended to an operand,
                           ; the corresponding integer is an
                           ; address rather than the actual operand
                           ; value.  This symbol is mandatory for
                           ; reference operands, optional for
                           ; multitypes and addresses, and
                           ; disallowed for literals.
label             =    ":" name
directive         =    padding / data / set / readonly /
                       unknown-directive
unknown-directive =    name ["(" expression *("," expression) ")"]
                           ; The parser can ignore unknown
                           ; directives.  The resulting bytecode
                           ; may or may not generate the expected
                           ; results.
padding           =    ("pad" / "align" / "at") "(" expression ")"
data              =    ("byte" / "word") "(" expression *("," 
                       expression) ")"
readonly          =    "readonly" "(" "0" / "1" ")"
set               =    "set" "(" name "," expression ")"
expression        =    value / "(" expression operator expression ")"
value             =    dec / bin / hex / name / "." / "!"
                           ; "." is the location of this
                           ; instruction/directive, whereas "!" is
                           ; the location of the closest
                           ; DECOMPRESSION-FAILURE
```

The following sections define how to convert the instructions, labels and directives into UDVM bytecode:

3.2.1. Expressions

The operand values needed by particular instructions or directives can be given in the form of expressions. An expression can include one or more values specified as decimal, binary, or hex (binary values are preceded by "0b" and hex values are preceded by "0x"). The expression may also include one or more of the following operators:

"+"	Addition
"-"	Subtraction
"*"	Multiplication
"/"	Integer division
"%"	Modulo arithmetic (a%b := a modulo b)
"&"	Binary AND
"|"	Binary OR
"^"	Binary XOR
"~"	Binary XNOR
"<<"	Binary LSHIFT
">>"	Binary RSHIFT

The operands for each operator must always be surrounded by parentheses so that the order in which the operators should be evaluated is clear. For example:

((1 + (2 * 3)) & (0xabcd - 0b00101010)) gives the result 3.

Expressions can also include the special values "." and "!". When the symbol "." is encountered, it is replaced by the location in the bytecode of the current instruction/directive. When the symbol "!" is encountered it is replaced by the location in the bytecode of the closest DECOMPRESSION-FAILURE instruction (i.e., the closest zero byte). This can be useful when writing UDVM instructions that call a decompression failure, for example:

INPUT-BYTES (1, temp, !)

The above instruction causes a decompression failure to occur if it tries to input data from beyond the end of the compressed message.

Note: When using "!" in the assembly language to generate bytecode, care must be taken to ensure that the address of the zero used at bytecode generation time will still contain zero when the bytecode is run. The readonly directive (see Section 3.2.3) can be used to do this.

It is also possible to assign integer values to text names: when a text name is encountered in an expression, it is replaced by the integer value assigned to it. Section 3.2.3 explains how to assign integer values to text names.

3.2.2. Instructions

A UDVM instruction is specified by the instruction opcode followed by zero or more operands. The instruction operands are enclosed in parentheses and separated by commas, for example:

```
ADD ($3, 4)
```

When generating the bytecode, the parser should replace the instruction opcode with the corresponding 1-byte value as per Figure 11 of SigComp [2].

Each operand consists of an expression that evaluates to an integer, optionally preceded by the symbol "$". This symbol indicates that the supplied integer value must be interpreted as the memory address at which the operand value can be found, rather than the actual operand value itself.

When converting each instruction operand to bytecode, the parser first determines whether the instruction expects the operand to be a literal, a reference, a multitype, or an address. If the operand is a literal, then, as per Figure 8 of SigComp, the parser inserts bytecode (usually the shortest) capable of encoding the supplied operand value.

Since literal operands are used to indicate the total number of operands for an instruction, it is possible to leave a literal operand blank and allow its value to be inferred automatically by the assembler. For example:

```
MULTILOAD (64, , 1, 2, 3, 4)
```

The missing operand should be given the value 4 because it is followed by a total of 4 operands.

If the operand is a reference, then, as per Figure 9 of SigComp, the parser inserts bytecode (usually the shortest) capable of encoding the supplied memory address. Note that reference operands will always be preceded by the symbol "$" in assembly because they always encode memory addresses rather than actual operand values.

If the operand is a multitype, then the parser first checks whether the symbol "$" is present. If so, then, as per Figure 10 of SigComp, it inserts bytecode (usually the shortest) capable of encoding the supplied integer as a memory address. If not, then, as per Figure 10 of SigComp, it inserts bytecode (usually the shortest) that encodes the supplied integer as an operand value.

If the operand is an address, then the parser checks whether the symbol "$" is present. If so, then the supplied integer is encoded as a memory address, just as for the multitype instruction above. If not, then the byte position of the opcode is subtracted from the supplied integer modulo 16, and the result is encoded as an operand value as per Figure 10 of SigComp.

The length of the resulting bytecode is dependent on the parser in use. There can be several correct and usable representations of the same instruction.

3.2.3. Directives

The assembly language provides a number of directives for evaluating expressions, moving instructions to a particular memory address, etc.

The directives "pad", "align", and "at" can be used to add padding to the bytecode.

The directive "pad (n)" appends n consecutive padding bytes to the bytecode. The actual value of the padding bytes is unimportant, so when the bytecode is uploaded to the UDVM, the padding bytes can be set to the initial values contained in the UDVM memory (this helps to reduce the size of a SigComp message).

The directive "align (n)" appends the minimum number of padding bytes to the bytecode such that the total number of bytes of bytecode generated so far is a multiple of n bytes. If the bytecode is already aligned to a multiple of n bytes, then no padding bytes are added.

The directive "at (n)" appends enough padding bytes to the bytecode such that the total number of bytes of bytecode generated so far is exactly n bytes. If more than n bytes have already been generated before the "at" directive is encountered then the assembly code contains an error.

The directives "byte" and "word" can be used to add specific data strings to the bytecode.

The directive "byte (n[0],..., n[k-1])" appends k consecutive bytes to the bytecode. The byte string is supplied as expressions that evaluate to give integers n[0],..., n[k-1] from 0 to 255.

The directive "word (n[0],..., n[k-1])" appends k consecutive 2-byte words to the bytecode. The word string is supplied as expressions that evaluate to give integers n[0],..., n[k-1] from 0 to 65535.

The directive "set (name, n)" assigns an integer value n to a specified text name. The integer value can be supplied in the form of an expression.

The directive "readonly (n)" where n is 0 or 1 can be used to indicate that an area of memory could be changed (0) or will not be changed (1) during the execution of the UDVM. This directive could be used, for example, in conjunction with "!" to ensure that the address of the zero used will still contain zero when the bytecode is executed. If no readonly directive is used, then any address containing zero can be used by "!" (i.e., by default, there is assumed to be a readonly (1) directive at Address 0) and it is up to the author of the assembly code to ensure that the address in question will still contain zero when the bytecode is executed. If the readonly directive is used, then bytes between a readonly (0) and readonly (1) pair are NOT to be used by "!". When a readonly directive has been used, the bytes obey that directive from that address to either another readonly directive or the end of UDVM memory, whichever comes first.

3.2.4. Labels

A label is a special directive used to assign memory addresses to text names.

Labels are specified by a single colon followed by the text name to be defined. The (absolute) position of the byte immediately following the label is evaluated and assigned to the text name. For example:

```
:start

LOAD (temp, 1)
```

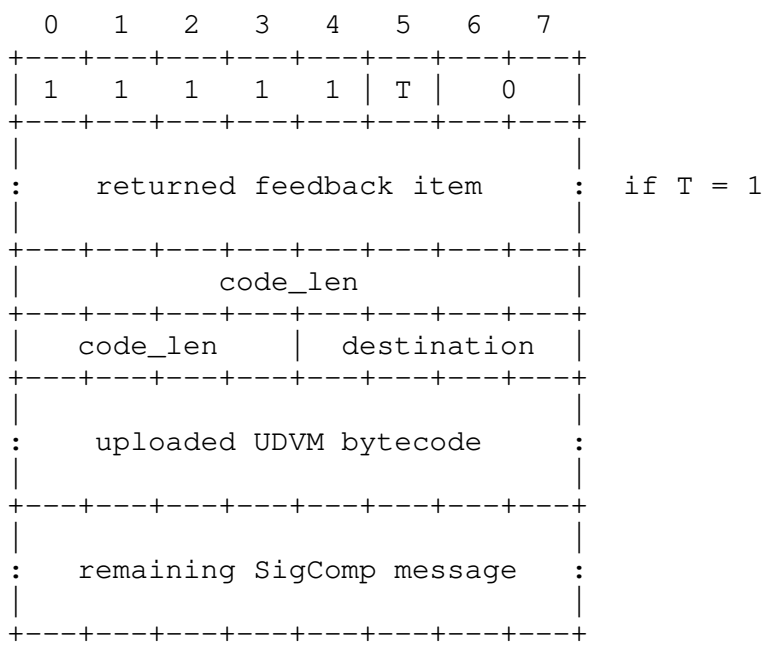
Since the label "start" occurs at the beginning of the bytecode, it is assigned the integer value 0.

Note that writing the label ":name" has exactly the same behavior as writing the directive "set (name, .)".

3.3. Uploading the Bytecode to the UDVM

Once the parser has converted a string of assembly into the corresponding bytecode, it must be copied to the UDVM memory beginning at Address 0 and then executed, beginning from the first UDVM instruction in the bytecode.

SigComp provides the following message format for uploading bytecode to the UDVM:



The destination field should be set to the memory address of the first UDVM instruction. Note that if this address cannot be represented by the destination field, then the bytecode cannot be uploaded to the UDVM using the standard SigComp header. In particular, the memory address of the first UDVM instruction must always be a multiple of 64 bytes or the standard SigComp header cannot be used. Of course, there may be other ways to upload the bytecode to the UDVM, such as retrieving the bytecode directly via the INPUT-BYTES instruction.

Additionally, all memory addresses between Address 0 and Address 31 inclusive are initialized to endpoint-specific values by the UDVM, so they must be specified as padding in the bytecode, or the standard SigComp header cannot be used. Memory addresses from Address 32 to Address (destination - 1) inclusive are initialized to 0, so they must be specified either as padding or as 0s if the bytecode is to be successfully uploaded using the standard SigComp header.

The code_len field should be set to the smallest value such that all memory addresses beginning at Address (destination + code_len) are either as initialised by the UDVM (to 0) or as set by the bytecode at runtime.

The "uploaded UDVM bytecode" should be set to contain the segment of bytecode that lies between Address (destination) and Address (destination + code_len - 1) inclusive.

4. Compression Algorithms

This section describes a number of compression algorithms that can be used by a SigComp compressor. In each case, the document provides UDVM bytecode for the corresponding decompression algorithm, which can be uploaded to the receiving endpoint as part of a SigComp message. Each algorithm (as written in this section) assumes that there is a 16K decompression memory size, there are 16 cycles per bit, and there is an 8K state memory size. Decompression will succeed with a smaller value for state memory size; however, the full state will not be created.

Section 4.1.1 covers a simple algorithm in some detail, including the steps required to compress and decompress a SigComp message. The remaining sections cover well-known compression algorithms that can be adapted for use in SigComp with minimal modification.

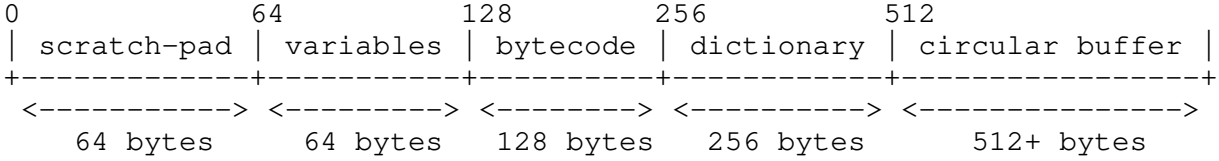
4.1. Well-known Compression Algorithms

4.1.1. LZ77

This section describes how to implement a very simple compression algorithm based on LZ77 [5].

A compressed message generated by the simplified LZ77 scheme consists of a sequence of 4-byte characters, where each character contains a 2-byte position value followed by a 2-byte length value. Each pair of integers identifies a byte string in the UDVM memory; when concatenated, these byte strings form the decompressed message.

When implementing a bytecode decompressor for the simplified LZ77 scheme, the UDVM memory is partitioned into five distinct areas, as shown below:



The first 128 bytes are used to hold the 2-byte variables needed by the LZ77 decompressor. Within this memory, the first 64 bytes are used as a scratch-pad, holding the 2-byte variables that can be discarded between SigComp messages. In contrast, the next 64 bytes (and in fact all of the UDVM memory starting from Address 64) should be saved after decompressing a SigComp message to improve the compression ratio of subsequent messages.

The bytecode for the LZ77 decompressor is stored beginning at Address 128. A total of 128 bytes are reserved for the bytecode although the LZ77 decompressor requires less; this allows room for adding additional features to the decompressor at a later stage.

The next 256 bytes are initialized by the bytecode to contain the integers 0 to 255 inclusive. The purpose of this memory area is to provide a dictionary of all possible uncompressed characters; this is important to ensure that the compressor can always generate a sequence of position/length pairs that encode a given message. For example, a byte with value 0x41 (corresponding to the ASCII character "A") can be found at Address 0x0141 of the UDVM memory, so the compressed character 0x0141 0001 will decompress to give this ASCII character. Note that encoding each byte in the application message as a separate 4-byte compressed character is not recommended, however, as the resulting "compressed" message is four times as large as the original uncompressed message.

The compression ratio of LZ77 is improved by the remaining UDVM memory, which is used to store a history buffer containing the previously decompressed messages. Compressed characters can point to strings that have previously been decompressed and stored in the buffer, so the overall compression ratio of the LZ77 algorithm improves as the decompressor "learns" more text strings and is able to encode longer strings using a single compressed character. The buffer is circular, so older messages are overwritten by new data when the buffer becomes full.

The steps required to implement an LZ77 compressor and decompressor are similar, although compression is more processor-intensive as it requires a searching operation to be performed. Assembly for the simplified LZ77 decompressor is given below:

```
; Variables that do not need to be stored after decompressing each
; SigComp message are stored here:

at (32)

:position_value                pad (2)
:length_value                  pad (2)
```

```
   at (42)

   set (requested_feedback_location, 0)

   ; The UDVM registers must be stored beginning at Address 64:

   at (64)

   ; Variables that should be stored after decompressing a message are
   ; stored here.  These variables will form part of the SigComp state
   ; item created by the bytecode:

   :byte_copy_left                 pad (2)
   :byte_copy_right                pad (2)
   :decompressed_pointer           pad (2)

   set (returned_parameters_location, 0)

   align (64)

   :initialize_memory

   set (udvm_memory_size, 8192)
   set (state_length, (udvm_memory_size - 64))

   ; The UDVM registers byte_copy_left and byte_copy_right are set to
   ; indicate the bounds of the circular buffer in the UDVM memory.  A
   ; variable decompressed_pointer is also created and set pointing to
   ; the start of the circular buffer:

   MULTILOAD (64, 3, circular_buffer, udvm_memory_size, circular_buffer)

   ; The "dictionary" area of the UDVM memory is initialized to contain
   ; the values 0 to 255 inclusive:

   MEMSET (static_dictionary, 256, 0, 1)

   :decompress_sigcomp_message

   :next_character

   ; The next character in the compressed message is read by the UDVM
   ; and the position and length integers are stored in the variables
   ; position_value and length_value, respectively.  If no more
   ; compressed data is available, the decompressor jumps to the
   ; "end_of_message" subroutine:

   INPUT-BYTES (4, position_value, end_of_message)
```

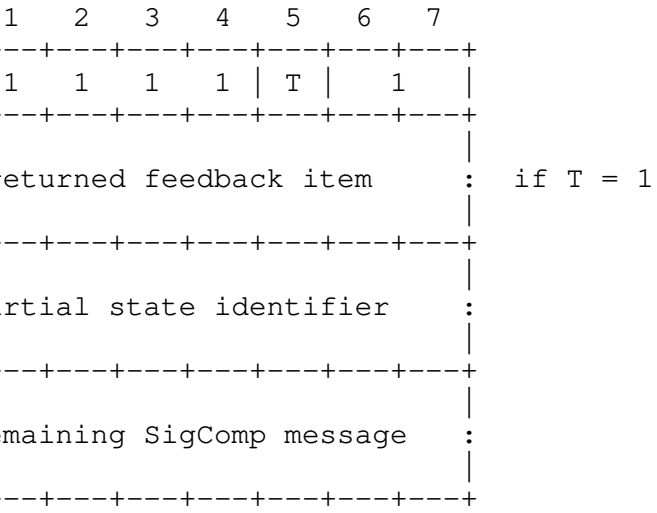
```
; The position_value and length_value point to a byte string in the
; UDVM memory, which is copied into the circular buffer at the
; position specified by decompressed_pointer.  This allows the string
; to be referenced by later characters in the compressed message:

COPY-LITERAL ($position_value, $length_value, $decompressed_pointer)

; The byte string is also outputted onto the end of the decompressed
; message:

OUTPUT ($position_value, $length_value)

; The decompressor jumps back to consider the next character in the
; compressed message:

JUMP (next_character)

:end_of_message

; The decompressor saves the UDVM memory and halts:

END-MESSAGE (requested_feedback_location,
returned_parameters_location, state_length, 64,
decompress_sigcomp_message, 6, 0)

at (256)

; Memory for the dictionary and the circular buffer are reserved by
; the following statements:

:static_dictionary              pad (256)
:circular_buffer
```

The task of an LZ77 compressor is simply to discover a sequence of 4-byte compressed characters that the above bytecode will decompress to give the desired application message. As an example, a message compressed using the simplified LZ77 algorithm is given below:

```
0x0154 0001 0168 0001 0165 0001 0120 0001 0152 0001 0165 0001 0173
0x0002 0161 0001 0175 0001 0172 0001 0161 0001 016e 0001 0174 0001
0x0120 0001 0161 0001 020d 0002 0174 0001 0201 0003 0145 0001 016e
0x0001 0164 0001 0120 0001 016f 0001 0166 0001 0211 0005 0155 0001
0x016e 0001 0169 0001 0176 0001 0165 0001 0172 0002 0165 0001 010a
0x0001
```

The uncompressed message is "The Restaurant at the End of the Universe\n".

The bytecode for the LZ77 decompressor can be uploaded as part of the compressed message, as specified in Section 3.3. However, in order to improve the overall compression ratio, it is important to avoid uploading bytecode in every compressed message. For this reason, SigComp allows the UDVM to save an area of its memory as a state item between compressed messages. Once a state item has been created, it can be retrieved by sending the corresponding state identifier using the following SigComp message format:



The partial_state_identifier field must contain the first 6 bytes of the state identifier for the state item to be accessed (see [2] for details of how state identifiers are derived).

Note that the partial_state_identifier field could be 9 or 12 bytes and that in these cases, bits 6 and 7 of the first byte of the message would be 10 or 11, respectively.

4.1.2. LZSS

This section provides UDVM bytecode for the simple but effective LZSS compression algorithm [6].

The principal improvement offered by LZSS over LZ77 is that each compressed character begins with a 1-bit indicator flag to specify whether the character is a literal or an offset/length pair. A literal value is simply a single uncompressed byte that is appended directly to the decompressed message.

An offset/length pair contains a 12-bit offset value from 1 to 4096 inclusive, followed by a 4-bit length value from 3 to 18 inclusive. Taken together, these values specify one of the previously received

text strings in the circular buffer, which is then appended to the end of the decompressed message.

Assembly for an LZSS decompressor is given below:

```
at (32)
readonly (0)

:index                          pad (2)
:length_value                   pad (2)
:old_pointer                    pad (2)

at (42)

set (requested_feedback_location, 0)

at (64)

:byte_copy_left                 pad (2)
:byte_copy_right                pad (2)
:input_bit_order                pad (2)
:decompressed_pointer           pad (2)

set (returned_parameters_location, 0)

align (64)
readonly (1)

:initialize_memory

set (udvm_memory_size, 8192)
set (state_length, (udvm_memory_size - 64))

MULTILOAD (64, 4, circular_buffer, udvm_memory_size, 0,
circular_buffer)

:decompress_sigcomp_message

:next_character

INPUT-HUFFMAN (index, end_of_message, 2, 9, 0, 255, 16384, 4, 4096,
8191, 1)
COMPARE ($index, 8192, length, end_of_message, literal)

:literal

set (index_lsb, (index + 1))
```

```
OUTPUT (index_lsb, 1)
COPY-LITERAL (index_lsb, 1, $decompressed_pointer)
JUMP (next_character)

:length

INPUT-BITS (4, length_value, !)
ADD ($length_value, 3)
LOAD (old_pointer, $decompressed_pointer)
COPY-OFFSET ($index, $length_value, $decompressed_pointer)
OUTPUT ($old_pointer, $length_value)
JUMP (next_character)

:end_of_message

END-MESSAGE (requested_feedback_location,
returned_parameters_location, state_length, 64,
decompress_sigcomp_message, 6, 0)

readonly (0)
:circular_buffer
```

An example of a message compressed using the LZSS algorithm is given below:

```
0x279a 0406 e378 b200 6074 1018 4ce6 1349 b842
```

The uncompressed message is "Oh no, not again!".

4.1.3. LZW

This section provides UDVM bytecode for the well-known LZW compression algorithm LZW [7]. This algorithm is used in a number of standards including the GIF image format.

LZW compression operates in a similar manner to LZ77 in that it maintains a circular buffer of previously received decompressed data, and each compressed character references exactly one byte string from the circular buffer. However, LZW also maintains a "codebook" containing 1024 position/length pairs that point to byte strings that LZW believes are most likely to occur in the uncompressed data.

The byte strings stored in the LZW codebook can be referenced by sending a single 10-bit value from 0 to 1023 inclusive. The UDVM extracts the corresponding text string from the codebook and appends it to the end of the decompressed message. It then creates a new codebook entry containing the current text string and the next character to occur in the decompressed message.

Assembly for an LZW decompressor is given below:

```
at (32)

:length_value                   pad (2)
:position_value                 pad (2)
:index                          pad (2)

at (42)

set (requested_feedback_location, 0)

at (64)

:byte_copy_left                 pad (2)
:byte_copy_right                pad (2)
:input_bit_order                pad (2)

:codebook_next                  pad (2)
:current_length                 pad (2)
:decompressed_pointer           pad (2)

set (returned_parameters_location, 0)

align (64)

:initialize_memory

set (udvm_memory_size, 8192)
set (state_length, (udvm_memory_size - 64))

MULTILOAD (64, 6, circular_buffer, udvm_memory_size, 0, codebook, 1,
static_dictionary)

:initialize_codebook

; The following instructions are used to initialize the first 256
; entries in the LZW codebook with single ASCII characters:

set (index_lsb, (index + 1))
set (current_length_lsb, (current_length + 1))

COPY-LITERAL (current_length_lsb, 3, $codebook_next)
COPY-LITERAL (index_lsb, 1, $decompressed_pointer)
ADD ($index, 1)
COMPARE ($index, 256, initialize_codebook, next_character, 0)

:decompress_sigcomp_message
```

```
   :next_character

   ; The following INPUT-BITS instruction extracts 10 bits from the
   ; compressed message:

   INPUT-BITS (10, index, end_of_message)

   ; The following instructions interpret the received bits as an index
   ; into the LZW codebook and extract the corresponding
   ; position/length pair:

   set (length_value_lsb, (length_value + 1))

   MULTIPLY ($index, 3)
   ADD ($index, codebook)
   COPY ($index, 3, length_value_lsb)

   ; The following instructions append the selected text string to the
   ; circular buffer and create a new codebook entry pointing to this
   ; text string:

   LOAD (current_length, 1)
   ADD ($current_length, $length_value)
   COPY-LITERAL (current_length_lsb, 3, $codebook_next)
   COPY-LITERAL ($position_value, $length_value, $decompressed_pointer)

   ; The following instruction outputs the text string specified by the
   ; position/length pair:

   OUTPUT ($position_value, $length_value)
   JUMP (next_character)

   :end_of_message

   END-MESSAGE (requested_feedback_location,
   returned_parameters_location, state_length, 64,
   decompress_sigcomp_message, 6, 0)

   :static_dictionary              pad (256)
   :circular_buffer

   at (4492)

   :codebook
```

An example of a message compressed using the LZW algorithm is given below:

0x14c6 f080 6c1b c6e1 9c20 1846 e190 201d 0684 206b 1cc2 0198 6f1c
0x9071 b06c 42c6 8195 111a 4731 a021 02bf f0

The uncompressed message is "So long and thanks for all the fish!\n".

4.1.4. DEFLATE

This section provides UDVM bytecode for the DEFLATE compression algorithm. DEFLATE is the algorithm used in the well-known "gzip" file format.

The following bytecode will decompress the DEFLATE compressed data format [8] with the following modifications:

1. The DEFLATE compressed data format separates blocks of compressed data by transmitting 7 consecutive zero bits. Each SigComp message is assumed to contain a separate block of compressed data, so the end-of-block bits are implicit and do not need to be transmitted at the end of a SigComp message.

2. This bytecode supports only DEFLATE block type 01 (data compressed with fixed Huffman codes).

Assembly for the DEFLATE decompressor is given below:

```
at (32)
readonly (0)

:index                          pad (2)
:extra_length_bits              pad (2)
:length_value                   pad (2)
:extra_distance_bits            pad (2)
:distance_value                 pad (2)

at (42)

set (requested_feedback_location, 0)

at (64)

:byte_copy_left                 pad (2)
:byte_copy_right                pad (2)
:input_bit_order                pad (2)
:decompressed_pointer           pad (2)
```

```
   :length_table                    pad (116)
   :distance_table                  pad (120)

   set (returned_parameters_location, 0)

   align (64)

   readonly (1)
   :initialize_memory

   set (udvm_memory_size, 8192)
   set (state_length, (udvm_memory_size - 64))
   set (length_table_start, (((length_table - 4) + 65536) / 4))
   set (length_table_mid, (length_table_start + 24))
   set (distance_table_start, (distance_table / 4))

   MULTILOAD (64, 122, circular_buffer, udvm_memory_size, 5,
   circular_buffer,

   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,

   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,       769,     9,       1025,
   9,       1537,    10,      2049,    10,      3073,
   11,      4097,    11,      6145,    12,      8193,
   12,      12289,   13,      16385,   13,      24577)

   :decompress_sigcomp_message

   INPUT-BITS (3, extra_length_bits, !)

   :next_character
```

```
   INPUT-HUFFMAN (index, end_of_message, 4,
       7, 0, 23, length_table_start,
       1, 48, 191, 0,
       0, 192, 199, length_table_mid,
       1, 400, 511, 144)
   COMPARE ($index, length_table_start, literal, end_of_message,
   length_distance)

   :literal

   set (index_lsb, (index + 1))

   OUTPUT (index_lsb, 1)
   COPY-LITERAL (index_lsb, 1, $decompressed_pointer)
   JUMP (next_character)

   :length_distance

   ; this is the length part

   MULTIPLY ($index, 4)
   COPY ($index, 4, extra_length_bits)
   INPUT-BITS ($extra_length_bits, extra_length_bits, !)
   ADD ($length_value, $extra_length_bits)

   ; this is the distance part

   INPUT-HUFFMAN (index, !, 1, 5, 0, 31, distance_table_start)
   MULTIPLY ($index, 4)
   COPY ($index, 4, extra_distance_bits)

   INPUT-BITS ($extra_distance_bits, extra_distance_bits, !)
   ADD ($distance_value, $extra_distance_bits)
   LOAD (index, $decompressed_pointer)
   COPY-OFFSET ($distance_value, $length_value, $decompressed_pointer)
   OUTPUT ($index, $length_value)
   JUMP (next_character)

   :end_of_message

   END-MESSAGE (requested_feedback_location,
   returned_parameters_location, state_length, 64,
   decompress_sigcomp_message, 6, 0)

   readonly (0)
   :circular_buffer
```

An example of a message compressed using the DEFLATE algorithm is given below:

0xf3c9 4c4b d551 28c9 4855 08cd cb2c 4b2d 2a4e 5548 cc4b 5170 0532
0x2b4b 3232 f3d2 b900

The uncompressed message is "Life, the Universe and Everything\n".

4.1.5. LZJH

This section provides UDVM bytecode for the LZJH compression algorithm. LZJH is the algorithm adopted by the International Telecommunication Union (ITU-T) Recommendation V.44 [9].

Assembly for the LZJH decompressor is given below:

```
at (32)
readonly (0)

; The following 2-byte variables are stored in the scratch-pad memory
; area because they do not need to be saved after decompressing a
; SigComp message:

:length_value                   pad (2)
:position_value                 pad (2)
:index                          pad (2)
:extra_extension_bits           pad (2)
:codebook_old                   pad (2)

at (42)

set (requested_feedback_location, 0)

at (64)

; UDVM_registers

:byte_copy_left                 pad (2)
:byte_copy_right                pad (2)

:input_bit_order                pad (2)
```

```
; The following 2-byte variables are saved as state after
; decompressing a SigComp message:

:current_length                 pad (2)
:decompressed_pointer           pad (2)
:ordinal_length                 pad (2)
:codeword_length                pad (2)
:codebook_next                  pad (2)

set (returned_parameters_location, 0)

align (64)
readonly (1)

:initialize_memory

; The following constants can be adjusted to configure the LZJH
; decompressor.  The current settings are as recommended in the V.44
; specification (given that a total of 8K UDVM memory is available):

set (udvm_memory_size, 8192)   ; sets the total memory for LZJH
set (max_extension_length, 8)  ; sets the maximum string extension
set (min_ordinal_length, 7)    ; sets the minimum ordinal length
set (min_codeword_length, 6)   ; sets the minimum codeword length

set (codebook_start, 4492)
set (first_codeword, (codebook_start - 12))
set (state_length, (udvm_memory_size - 64))

MULTILOAD (64, 8, circular_buffer, udvm_memory_size, 7, 0,
circular_buffer, min_ordinal_length, min_codeword_length,
codebook_start)

:decompress_sigcomp_message

:standard_prefix

; The following code decompresses the standard 1-bit LZJH prefix
; that specifies whether the next character is an ordinal or a
; codeword/control value:

INPUT-BITS (1, index, end_of_message)
COMPARE ($index, 1, ordinal, codeword_control, codeword_control)

:prefix_after_codeword
```

```
; The following code decompresses the special LZJH prefix that only
; occurs after a codeword.  It specifies whether the next character
; is an ordinal, a codeword/control value, or a string extension:

INPUT-HUFFMAN (index, end_of_message, 2, 1, 1, 1, 2, 1, 0, 1, 0)
COMPARE ($index, 1, ordinal, string_extension, codeword_control)

:ordinal

; The following code decompresses an ordinal character and creates
; a new codebook entry consisting of the ordinal character and the
; next character to be decompressed:

set (index_lsb, (index + 1))
set (current_length_lsb, (current_length + 1))

INPUT-BITS ($ordinal_length, index, !)
OUTPUT (index_lsb, 1)
LOAD (current_length, 2)
COPY-LITERAL (current_length_lsb, 3, $codebook_next)
COPY-LITERAL (index_lsb, 1, $decompressed_pointer)
JUMP (standard_prefix)

:codeword_control

; The following code decompresses a codeword/control value:

INPUT-BITS ($codeword_length, index, !)
COMPARE ($index, 3, control_code, initialize_memory, codeword)

:codeword

; The following code interprets a codeword as an index into the LZJH
; codebook.  It extracts the position/length pair from the specified
; codebook entry; the position/length pair points to a byte string
; in the circular buffer, which is then copied to the end of the
; decompressed message.  The code also creates a new codebook entry
; consisting of the byte string plus the next character to be
; decompressed:

set (length_value_lsb, (length_value + 1))

MULTIPLY ($index, 3)
ADD ($index, first_codeword)
COPY ($index, 3, length_value_lsb)
LOAD (current_length, 1)
ADD ($current_length, $length_value)
LOAD (codebook_old, $codebook_next)
```

```
   COPY-LITERAL (current_length_lsb, 3, $codebook_next)
   COPY-LITERAL ($position_value, $length_value, $decompressed_pointer)
   OUTPUT ($position_value, $length_value)
   JUMP (prefix_after_codeword)

   :string_extension

   ; The following code decompresses a Huffman-encoded string extension:

   INPUT-HUFFMAN (index, !, 4, 1, 1, 1, 1, 2, 1, 3, 2, 1, 1, 1, 13, 3,
   0, 7, 5)
   COMPARE ($index, 13, continue, extra_bits, extra_bits)

   :extra_bits

   INPUT-BITS (max_extension_length, extra_extension_bits, !)
   ADD ($index, $extra_extension_bits)

   :continue

   ; The following code extends the most recently created codebook entry
   ; by the number of bits specified in the string extension:

   COPY-LITERAL ($position_value, $length_value, $position_value)
   COPY-LITERAL ($position_value, $index, $decompressed_pointer)
   OUTPUT ($position_value, $index)
   ADD ($index, $length_value)
   COPY (index_lsb, 1, $codebook_old)
   JUMP (standard_prefix)

   :control_code

   ; The code can handle all of the control characters in V.44 except
   ; for ETM (Enter Transparent Mode), which is not required for
   ; message-based protocols such as SigComp.

   COMPARE ($index, 1, !, flush, stepup)

   :flush

   ; The FLUSH control character jumps to the beginning of the next
   ; complete byte in the compressed message:

   INPUT-BYTES (0, 0, 0)
   JUMP (standard_prefix)

   :stepup
```

```
; The STEPUP control character increases the number of bits used to
; encode an ordinal value or a codeword:

INPUT-BITS (1, index, !)
COMPARE ($index, 1, stepup_ordinal, stepup_codeword, 0)

:stepup_ordinal

ADD ($ordinal_length, 1)
JUMP (ordinal)

:stepup_codeword

ADD ($codeword_length, 1)
JUMP (codeword_control)

:end_of_message

END-MESSAGE (requested_feedback_location,
returned_parameters_location, state_length, 64,
decompress_sigcomp_message, 6, 0)

readonly (0)
:circular_buffer
```

An example of a message compressed using the LZJH algorithm is given below:

```
0x5c09 e6e0 cadc c8d2 dcce 40c2 40f2 cac2 e440 c825 c840 ccde 29e8
0xc2f0 40e0 eae4 e0de e6ca e65c 1403
```

The uncompressed message is "...spending a year dead for tax purposes.\n".

4.2. Adapted Algorithms

4.2.1. Modified DEFLATE

Alternative algorithms can also be used with SigComp. This section shows a modified version of the DEFLATE [8] algorithm. The two-stage encoding of DEFLATE is replaced by a single step with a discrete Huffman code for each symbol. The literal/length symbol probabilities are dependent upon whether the previous symbol was a literal or a match. Bit handling is also simpler, in that all bits are input using the INPUT-HUFFMAN instruction and the value of the H bit does not change so all bits are input, read, and interpreted in the same order.

Assembly for the algorithm is given below. String matching rules are the same as for the other LZ-based algorithms, with the alternative encoding of the literals and length/distance pairs.

```
at (32)
readonly (0)

:index                          pad (2)
:distance_value                 pad (2)
:old_pointer                    pad (2)

at (42)

set (requested_feedback_location, 0)

at (64)

:byte_copy_left                 pad (2)
:byte_copy_right                pad (2)
:input_bit_order                pad (2)
:decompressed_pointer           pad (2)

set (returned_parameters_location, 0)

at (128)
readonly (1)

:initialize_memory

set (udvm_memory_size, 8192)
set (state_length, (udvm_memory_size - 64))

MULTILOAD (64, 4, circular_buffer, udvm_memory_size, 0,
circular_buffer)

:decompress_sigcomp_message

:character_after_literal

INPUT-HUFFMAN (index, end_of_message, 16,
    5, 0, 11, 46,
    0, 12, 12, 256,
    1, 26, 32, 257,
    1, 66, 68, 32,
    0, 69, 94, 97,
    0, 95, 102, 264,
    0, 103, 103, 511,
    2, 416, 426, 35,
```

```
        0, 427, 465, 58,
        0, 466, 481, 272,
        1, 964, 995, 288,
        3, 7968, 7988, 123,
        0, 7989, 8115, 384,
        1, 16232, 16263, 0,
        0, 16264, 16327, 320,
        1, 32656, 32767, 144)

   COMPARE ($index, 256, literal, distance, distance)

   :character_after_match

   INPUT-HUFFMAN (index, end_of_message, 16,
        4, 0, 0, 511,
        1, 2, 9, 256,
        1, 20, 22, 32,
        0, 23, 30, 264,
        1, 62, 73, 46,
        0, 74, 89, 272,
        2, 360, 385, 97,
        0, 386, 417, 288,
        1, 836, 874, 58,
        0, 875, 938, 320,
        1, 1878, 1888, 35,
        0, 1889, 2015, 384,
        1, 4032, 4052, 123,
        1, 8106, 8137, 0,
        1, 16276, 16379, 144,
        1, 32760, 32767, 248)

   COMPARE ($index, 256, literal, distance, distance)

   :literal

   set (index_lsb, (index + 1))

   OUTPUT (index_lsb, 1)
   COPY-LITERAL (index_lsb, 1, $decompressed_pointer)
   JUMP (character_after_literal)

   :distance

   SUBTRACT ($index, 253)
   INPUT-HUFFMAN (distance_value, !, 9,
        9, 0, 7, 9,
        0, 8, 63, 129,
        1, 128, 135, 1,
```

```
        0, 136, 247, 17,
        0, 248, 319, 185,
        1, 640, 1407, 257,
        2, 5632, 6655, 1025,
        1, 13312, 15359, 2049,
        2, 61440, 65535, 4097)

   LOAD (old_pointer, $decompressed_pointer)
   COPY-OFFSET ($distance_value, $index, $decompressed_pointer)
   OUTPUT ($old_pointer, $index)
   JUMP (character_after_match)

   :end_of_message

   END-MESSAGE (requested_feedback_location,
   returned_parameters_location, state_length, 64,
   decompress_sigcomp_message, 6, 0)

   readonly (0)
   :circular_buffer
```

An example of a message compressed using the modified DEFLATE algorithm is given below:

```
0xd956 b132 cd68 5424 c5a9 6215 8a70 a64d af0a 5499 3621 509b 3e4c
0x28b4 a145 b362 653a d0a6 498b 5a6d 2970 ac4c 930a a4ca 74a4 c268
0x0c
```

The uncompressed message is "Arthur leapt to his feet like an author hearing the phone ring".

5. Additional SigComp Mechanisms

This section covers the additional mechanisms that can be employed by SigComp to improve the overall compression ratio, including the use of acknowledgements, dictionaries, and sharing state between two directions of a compressed message flow.

An example of assembly code is provided for these mechanisms. Depending on the mechanism and basic algorithm in use, the assembly code for either the mechanism or the basic algorithm may require modification (e.g., if the algorithm uses 'no more input' to jump to end_of_message, following end_of_message with an input instruction for CRC will not work). In any case, these are examples and there may be alternative ways to make use of the mechanisms.

When each of the compression algorithms described in Section 4 has successfully decompressed the current SigComp message, the contents of the UDVM memory are saved as a SigComp state item. Subsequent messages can access this state item by uploading the correct state identifier to the receiving endpoint, which avoids the need to upload the bytecode for the compression algorithm on a per-message basis. However, before a state item can be accessed, the compressor must first ensure that it is available at the receiving endpoint.

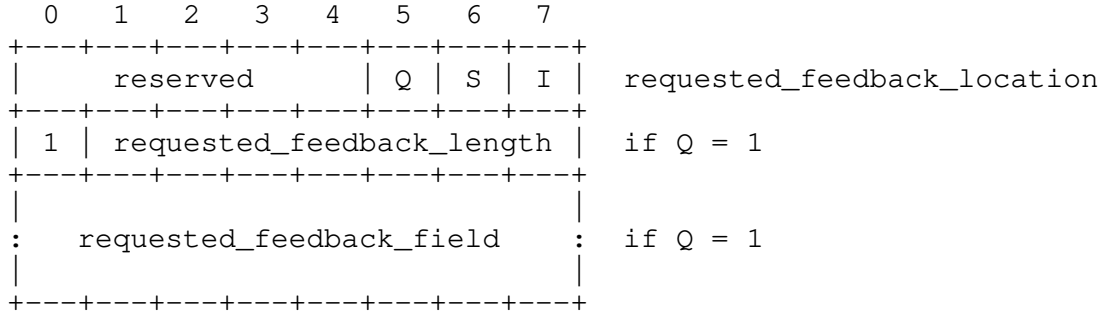
For each SigComp compartment, the receiving endpoint maintains a list of currently available states (where the total amount of state saved does not exceed the state_memory_size for the compartment). The SigComp compressor should maintain a similar list containing the states that it has instructed the receiving endpoint to save.

As well as tracking the list of state items that it has saved at the remote endpoint, the compressor also maintains a flag for each state item indicating whether or not the state can safely be accessed. State items should not be accessed until they have been acknowledged (e.g., by using the SigComp feedback mechanism as per Section 5.1).

State items are deleted from the list when adding a new piece of state when the total state_memory_size for the compartment is full. The state to be deleted is determined according to age and retention priority as discussed in SigComp [2]. The SigComp compressor should not attempt to access any state items that have been deleted in this manner, as they may no longer be available at the receiving endpoint.

5.1. Acknowledging a State Item

SigComp [2] defines a feedback mechanism to allow the compressor to request feedback from the decompressor, to give the compressor indication that a message has been received and correctly decompressed and that state storage has been attempted. (Note: This mechanism cannot convey the success or failure of individual state creation requests.) In order to invoke the feedback mechanism, the following fields must be reserved in the UDVM memory:



These fields can be reserved in any of the algorithms of Section 4 by replacing the line "set (requested_feedback_location, 0)" with the following assembly:

```
:requested_feedback_location    pad (1)
:requested_feedback_length      pad (1)
:requested_feedback_field       pad (12)
:hash_start                     pad (8)
```

When a SigComp message is successfully decompressed and saved as state, the following bytecode instructs the receiving endpoint to return the first 6 bytes of the corresponding state identifier. The bytecode can be added to any of the compression algorithms of Section 4 immediately following the ":end_of_message" label:

```
:end_of_message

set (hash_length, (state_length + 8))

LOAD (requested_feedback_location, 1158)
MULTILOAD (hash_start, 4, state_length, 64,
decompress_sigcomp_message, 6)
SHA-1 (hash_start, hash_length, requested_feedback_field)
```

The receiving endpoint then returns the state identifier in the "returned feedback field" of the next SigComp message to be transmitted in the reverse direction.

When the state identifier is returned, the compressor can set the availability flag for the corresponding state to 1.

5.2. Static Dictionary

Certain protocols that can be compressed using SigComp offer a fixed, mandatory state item known as a static dictionary. This dictionary contains a number of text strings that commonly occur in messages generated by the protocol in question. The overall compression ratio can often be improved by accessing the text phrases from this static dictionary rather than by uploading them as part of the compressed message.

As an example, a static dictionary is provided for the protocols SIP and SDP, RFC 3485 [4]. This dictionary is designed for use by a wide range of compression algorithms including all of the ones covered in Section 4.

In any of the compression algorithms of Section 4, the static dictionary can be accessed by inserting the following instruction immediately after the ":initialize_memory" label:

```
STATE-ACCESS (dictionary_id, 6, 0, 0, 1024, 0)
```

The parameters of STATE-ACCESS instruction will depend on the compression algorithm in use.

The following lines should also be inserted immediately after the END-MESSAGE instruction:

```
:dictionary_id

byte (0xfb, 0xe5, 0x07, 0xdf, 0xe5, 0xe6)
```

The text strings contained in the static dictionary can then be accessed in exactly the same manner as the text strings from previously decompressed messages (see Section 5.1 for further details).

Note that in some cases it is sufficient to load only part of the static dictionary into the UDVM memory. Further information on the contents of the SIP and SDP static dictionary can be found in the relevant document, RFC 3485 [4].

5.3. CRC Checksum

The acknowledgement scheme of Section 5.1 is designed to indicate the successful decompression of a message. However, it does not guarantee that the decompressed message is identical to the original message, since decompression of a corrupted message could succeed but with some characters being incorrect. This could lead to an incorrect message being passed to the application or unexpected contents of state to be stored. In order to prevent this happening, a CRC check could be used.

If an additional CRC check is required, then the following bytecode can be inserted after the ":end_of_message" label:

```
INPUT-BYTES (2, index, !)
CRC ($index, 64, state_length, !)
```

The bytecode extracts a 2-byte CRC from the end of the SigComp message and compares it with a CRC calculated over the UDVM memory. Decompression failure occurs if the two CRC values do not match.

A definition of the CRC polynomial used by the CRC instruction can be found in SigComp [2].

5.4. Announcing Additional Resources

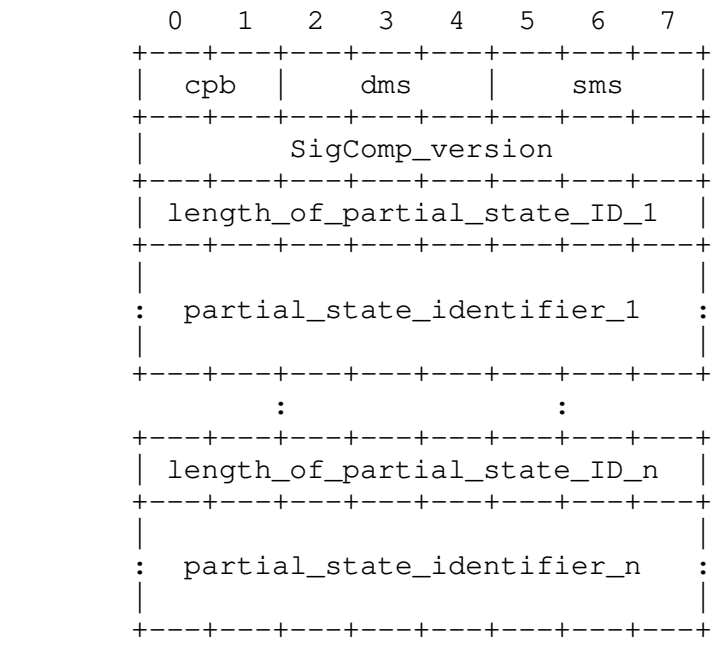
If a particular endpoint is able to offer more processing or memory resources than the mandatory minimum, the SigComp feedback mechanism can be used to announce that these resources are available to the remote endpoint. This may help to improve the overall compression ratio between the two endpoints.

Additionally, if an endpoint has any pieces of state that may be useful for the remote endpoint to reference, it can advertise the identifiers for the states. The remote endpoint can then make use of any that it also knows about (i.e., knows the contents of), for example, a dictionary or shared mode state (see Section 5.5).

The values of the following SigComp parameters can be announced using the SigComp advertisement mechanism:

- cycles_per_bit
- decompression_memory_size
- state_memory_size
- SigComp_version
- state identifiers

As explained in SigComp, in order to announce the values of these parameters, the following fields must be reserved in the UDVM memory:



These fields can be reserved in any of the algorithms of Section 4 by replacing the line "set (returned_parameters_location, 0)" with the following piece of assembly:

```
:adverts_len                    pad (1)
:adverts_len_lsb                pad (1)
:returned_parameters_location   pad (1)
:returned_sigcomp_version       pad (1)
:state_ids                      pad (x)
```

where x is enough space for the number state identifiers that the endpoint wishes to advertise.

When a SigComp message is successfully decompressed and saved as state, the following bytecode announces to the receiving endpoint that additional resources and pieces of state are available at the sending endpoint:

```
:end_of_message

LOAD (returned_parameters_location, N)
INPUT-BYTES (1, adverts_len_lsb, done)
INPUT-BYTES ($adverts_len, state_ids, done)
```

```
:done
```

Note that the integer value "N" should be set equal to the amount of resources available at the sending endpoint. N should be expressed as a 2-byte integer with the most significant bits corresponding to the cycles_per_bit parameter and the least significant bits corresponding to the SigComp_version parameter.

The length of the state identifiers followed by the state identifiers in the format shown are appended to the end of the compressed message.

5.5. Shared Compression

This section provides bytecode for implementing the SigComp shared compression mechanism, RFC 3321 [3]. If two endpoints A and B are communicating via SigComp, shared compression allows the messages sent from Endpoint A to Endpoint B to be compressed relative to the messages sent from Endpoint B to Endpoint A (and vice versa). This may improve the overall compression ratio by reducing the need to transmit the same information in both directions.

As described in RFC 3321 [3], two steps must be taken to implement shared compression at an endpoint.

First, it is necessary to announce to the remote endpoint that shared compression is available. This is done by announcing the state identifier as an available piece of state. This can be done using the returned_parameters_location announcement as in Section 5.4.

Second, assuming that such an announcement is received from the remote endpoint, then the state created by shared compression needs to be accessed by the message sent in the opposite direction. This can be done in a similar way to accessing the static dictionary (see Section 5.2), but using the appropriate state identifier, for example, by using the INPUT-BYTES instruction as below:

```
:shared_state_id        pad (6)

:access_shared_state

INPUT-BYTES (6, shared_state_id, !)
STATE-ACCESS (shared_state_id, 6, 0, 0, $decompressed_start, 0)
```

6. Security Considerations

This document describes implementation options for the SigComp protocol [2]. Consequently, the security considerations for this document match those of SigComp.

7. Acknowledgements

Thanks to Richard Price, Carsten Bormann, Adam Roach, Lawrence Conroy, Christian Schmidt, Max Riegel, Lars-Erik Jonsson, Jonathan Rosenberg, Stefan Forsgren, Krister Svanbro, Miguel Garcia, Christopher Clanton, Khiem Le, Ka Cheong Leung, and Zoltan Barczikay for valuable input and review.

Special thanks to Pekka Pessi and Cristian Constantin, who served as committed working group document reviewers.

8. Intellectual Property Right Considerations

9. Normative References

[1] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005.

[2] Price, R., Bormann, C., Christoffersson, J., Hannu, H., Liu, Z., and J. Rosenberg, "Signaling Compression (SigComp)", RFC 3320, January 2003.

[3] Hannu, H., Christoffersson, J., Forsgren, S., Leung, K.-C., Liu, Z., and R. Price, "Signaling Compression (SigComp) - Extended Operations", RFC 3321, January 2003.

[4] Garcia-Martin, M., Bormann, C., Ott, J., Price, R., and A.B. Roach, "The Session Initiation Protocol (SIP) and Session Description Protocol (SDP) Static Dictionary for Signaling Compression (SigComp)", RFC 3485, February 2003.

[5] Ziv, J. and A. Lempel, "A universal algorithm for sequential data compression", IEEE 23:337-343, 1977.

[6] Storer, J., "Data Compression: Methods and Theory", Computer Science Press ISBN 0-88175-161-8, 1998.

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

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

[9] "Data Compression Procedures", ITU-T Recommendation V.44, November 2000.

Appendix A. UDVM Bytecode for the Compression Algorithms

The following sections list the UDVM bytecode generated for each compression algorithm of Section 4.

Note that the different assemblers can output different bytecode for the same piece of assembly code, so a valid assembler can produce results different from those presented below. However, the following bytecode should always generate the same decompressed messages on any UDVM.

A.1. Well-known Algorithms

A.1.1. LZ77

```
0x0f86 0389 8d89 1588 8800 011c 0420 0d13 5051 2222 5051 16f5 2300
0x00bf c086 a08b 06
```

A.1.2. LZSS

```
0x0f86 04a0 c48d 00a0 c41e 2031 0209 00a0 ff8e 048c bfff 0117 508d
0x0f23 0622 2101 1321 0123 16e5 1d04 22e8 0611 030e 2463 1450 5123
0x2252 5116 9fd2 2300 00bf c086 a089 06
```

A.1.3. LZW

```
0x0f86 06a1 ce8d 00b1 8f01 a0ce 13a0 4903 2313 2501 2506 1201 1752
0x88f4 079f 681d 0a24 2508 1203 0612 b18f 1252 0321 0ea0 4801 0624
0x5013 a049 0323 1351 5025 2251 5016 9fde 2300 00bf c086 a09f 06
```

A.1.4. DEFLATE

```
0x0f86 7aa2 528d 05a2 5200 0300 0400 0500 0600 0700 0800 0900 0a01
0x0b01 0d01 0f01 1102 1302 1702 1b02 1f03 2303 2b03 3303 3b04 a043
0x04a0 5304 a063 04a0 7305 a083 05a0 a305 a0c3 05a0 e300 a102 0001
0x0002 0003 0004 0105 0107 0209 020d 0311 0319 0421 0431 05a0 4105
0xa061 06a0 8106 a0c1 07a1 0107 a181 08a2 0108 a301 09a4 0109 a601
0x0aa8 010a ac01 0bb0 010b b801 0c80 2001 0c80 3001 0d80 4001 0d80
0x6001 1d03 229f b41e 20a0 6504 0700 1780 4011 0130 a0bf 0000 a0c0
0xa0c7 8040 2901 a190 a1ff a090 1750 8040 1109 a046 1322 2101 1321
0x0123 169f d108 1004 1250 0422 1d51 229f d706 1251 1e20 9fcf 0105
0x001f 2f08 1004 1250 0426 1d53 26f6 0614 530e 2063 1454 5223 2250
0x5216 9f9e 2300 00bf c086 a1de 06
```

A.1.5. LZJH

```
0x0f86 08a1 5b8d 0700 a15b 0706 b18f 1d01 24a0 c317 5201 1a31 311e
0x24a0 b802 0101 0102 0100 0100 1752 0107 a04e 1e1d 6524 f822 2501
0x0ea0 4602 13a0 4703 2713 2501 2416 9fcd 1d66 24e1 1752 03a0 639f
0xb808 0812 0306 12b1 8312 5203 210e a046 0106 2350 0e28 6713 a047
0x0327 1351 5024 2251 5016 9fa8 1e24 9fb1 0401 0101 0102 0103 0201
0x0101 0d03 0007 0517 520d 0d06 061d 0826 f706 1253 1351 5011 1351
0x5224 2251 5206 1250 1225 0154 169f 6617 5201 9fdb 070f 1c00 009e
0xce16 9f57 1d01 24fa 1752 0107 0d9e c206 2501 169f 6506 2601 169f
0x7623 0000 bfc0 86a0 8e06
```

A.2. Adapted Algorithms

A.2.1. Modified DEFLATE

```
0x0f86 04a1 d38d 00a1 d31e 20a1 4010 0500 0b2e 000c 0c88 011a 20a1
0x0101 a042 a044 2000 a045 a05e a061 00a0 5fa0 66a1 0800 a067 a067
0xa1ff 02a1 a0a1 aa23 00a1 aba1 d13a 00a1 d2a1 e1a1 1001 a3c4 a3e3
0xa120 03bf 20bf 34a0 7b00 bf35 bfb3 a180 0180 3f68 803f 8700 0080
0x3f88 803f c7a1 4001 807f 9080 7fff a090 1750 88a0 79a0 83a0 831e
0x20a0 c810 0400 00a1 ff01 0209 8801 1416 2000 171e a108 013e a049
0x2e00 a04a a059 a110 02a1 68a1 81a0 6100 a182 a1a1 a120 01a3 44a3
0x6a3a 00a3 6ba3 aaa1 4001 a756 a760 2300 a761 a7df a180 01af c0af
0xd4a0 7b01 bfaa bfc9 0001 803f 9480 3ffb a090 0180 7ff8 807f ffa0
0xf817 5088 0610 1022 2101 1321 0123 169f 1107 10a0 fd1e 229f d909
0x0900 0709 0008 3fa0 8101 87a0 8701 00a0 88a0 f711 00a0 f8a1 3fa0
0xb901 a280 a57f a101 02b6 00b9 ffa4 0101 8034 0080 3bff a801 0290
0x00ff b001 0e24 6314 5150 2322 5250 169f 3b23 0000 bfc0 86a0 8906
```

Authors' Addresses

Abigail Surtees
Siemens/Roke Manor Research
Roke Manor Research Ltd.
Romsey, Hants SO51 0ZN
UK

Phone: +44 (0)1794 833131
EMail: abigail.surtees@roke.co.uk
URI: http://www.roke.co.uk

Mark A. West
Siemens/Roke Manor Research
Roke Manor Research Ltd.
Romsey, Hants SO51 0ZN
UK

Phone: +44 (0)1794 833311
EMail: mark.a.west@roke.co.uk
URI: http://www.roke.co.uk

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).