Network Working Group
Internet-Draft
Expires: August 2, 2002

R. Price
R. Hancock
S. McCann
A. Surtees
P. Ollis
M. West
Siemens/Roke Manor

February 2002

Framework for EPIC-LITE
draft-ietf-rohc-epic-lite-01.txt

Status of this Memo

Copyright Notice

Abstract

This draft describes the framework of the Efficient Protocol Independent Compression (EPIC-LITE) scheme.

The RObust Header Compression ROHC [1] scheme is designed to compress packet headers over error prone channels. It is built around an extensible core framework that can be tailored to compress new protocol stacks by adding additional ROHC profiles.

EPIC-LITE extends the basic ROHC framework by introducing a BNF-based input language that simplifies the creation of new ROHC [1] profiles.

Table of Contents

1. Introduction

This document describes a plug-in extension for the ROHC [1] framework which simplifies the creation of new compression profiles.

The Efficient Protocol Independent Compression (EPIC-LITE) scheme for generating new ROHC profiles takes as its input a choice of one or more compression techniques for each field in the protocol stack to be compressed. Using this input EPIC-LITE derives a set of compressed header formats that can be used to quickly and efficiently compress and decompress headers.

Chapter 2 explains some of the terminology used in the draft.

Chapter 3 gives an overview of the EPIC-LITE scheme.

Chapter 4 describes the language used by EPIC-LITE to create new profiles.

Chapter 5 considers the basic techniques available in the EPIC-LITE library of compression routines.

Chapter 6 specifies the parameters used to define a ROHC [1] profile.

Appendix A gives a normative description of EPIC-LITE in pseudocode.

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 [5].

Profile

A ROHC [1] profile is a description of how to compress a certain protocol stack over a certain type of link. Each profile includes one or more sets of compressed header formats and a state machine to control the compressor and the decompressor.

Context

The context is memory which stores one or more previous values of fields in the uncompressed header. The compressor and decompressor both maintain a copy of the context, and fields can be compressed relative to their stored values for better

compression efficiency.

Compressed header format

A compressed header format describes how to compress each field in the chosen protocol stack. It consists of two parts: a bit pattern to indicate to the decompressor which format is being used, followed by a list of the compressed versions of each field.

Encoding method

An encoding method is a procedure for compressing fields. Examples include STATIC encoding (field is the same as the context), INFERRED-OFFSET encoding (field is calculated at the

decompressor) and IRREGULAR encoding (field must be transmitted in full).

Indicator flags

Each EPIC-LITE compressed packet contains a set of indicator flags. The flags are placed at the front of the packet as a single bit pattern, and indicate to the decompressor exactly which encoding method has been applied to which field.

Set of compressed header formats

A complete set of compressed header formats uses up all of the indicator bit patterns available at the start of the compressed header. A profile may have several sets of compressed header formats available, but only one set can be in use at a given time.

Library of encoding methods

The library of encoding methods contains a number of commonly used procedures that can be called to compress fields in the chosen protocol stack. More encoding methods can be added to the library if they are needed.

BNF (Backus Naur Form)

BNF is a "metasyntax" commonly used to describe the syntax of protocols and languages.

BNF input language

EPIC-LITE describes a new ROHC profile using a simple BNF-based input language. The BNF description of the ROHC profile assigns one or more encoding methods to each field in the chosen stack.

Control data

The term 'control data' refers to any data passed between the compressor and decompressor, that is not part of the uncompressed header. An example of control data is the header checksum calculated by EPIC-LITE over the uncompressed header to ensure robustness against bit errors and dropped packets.

3. The EPIC-LITE framework for generating new ROHC profiles

This chapter outlines the EPIC-LITE framework for the creation of new ROHC profiles.

3.1 Structure of the EPIC-LITE compressed headers

Each compressed header is divided into two distinct parts: the indicator flags and the compressed fields as illustrated below:

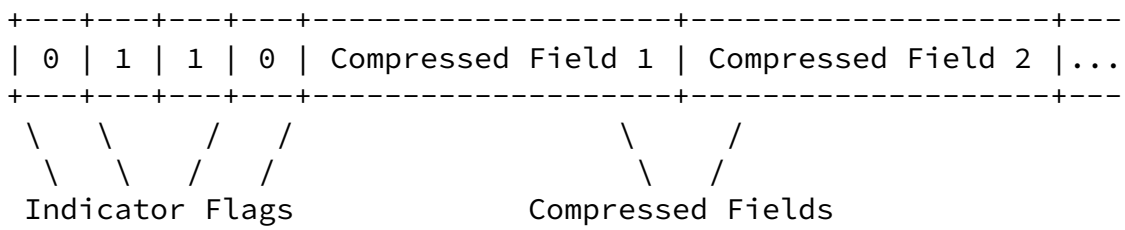

Figure 1 : Structure of an EPIC-LITE compressed header

The indicator flags specify how every field in the uncompressed header has been compressed, whilst the compressed fields contain enough information to transmit each field from the compressor to the decompressor. This information might be the entire uncompressed field, it might be LSBs (Least Significant Bits) of the uncompressed field etc.

Note that for simplicity EPIC-LITE always places the indicator flags at the front of the compressed header followed by each complete compressed field in turn. As for RFC-1144 [3] the compressed fields are in reverse order compared to the uncompressed header (this is a useful trick to speed up parsing at the decompressor).

Unlike other compression schemes, the header formats used by EPIC-LITE are not designed by hand but instead are generated automatically using a special algorithm. This means that EPIC-LITE can be applied to any protocol stack provided that it has been correctly programmed using the BNF-based input language described in Section 3.3.

3.2 Compression and decompression procedures

Figure 2 illustrates the processing which is done by EPIC-LITE for each header to be compressed and decompressed. Note that references are given to pseudocode in Appendix A which describes each of the stages explicitly.

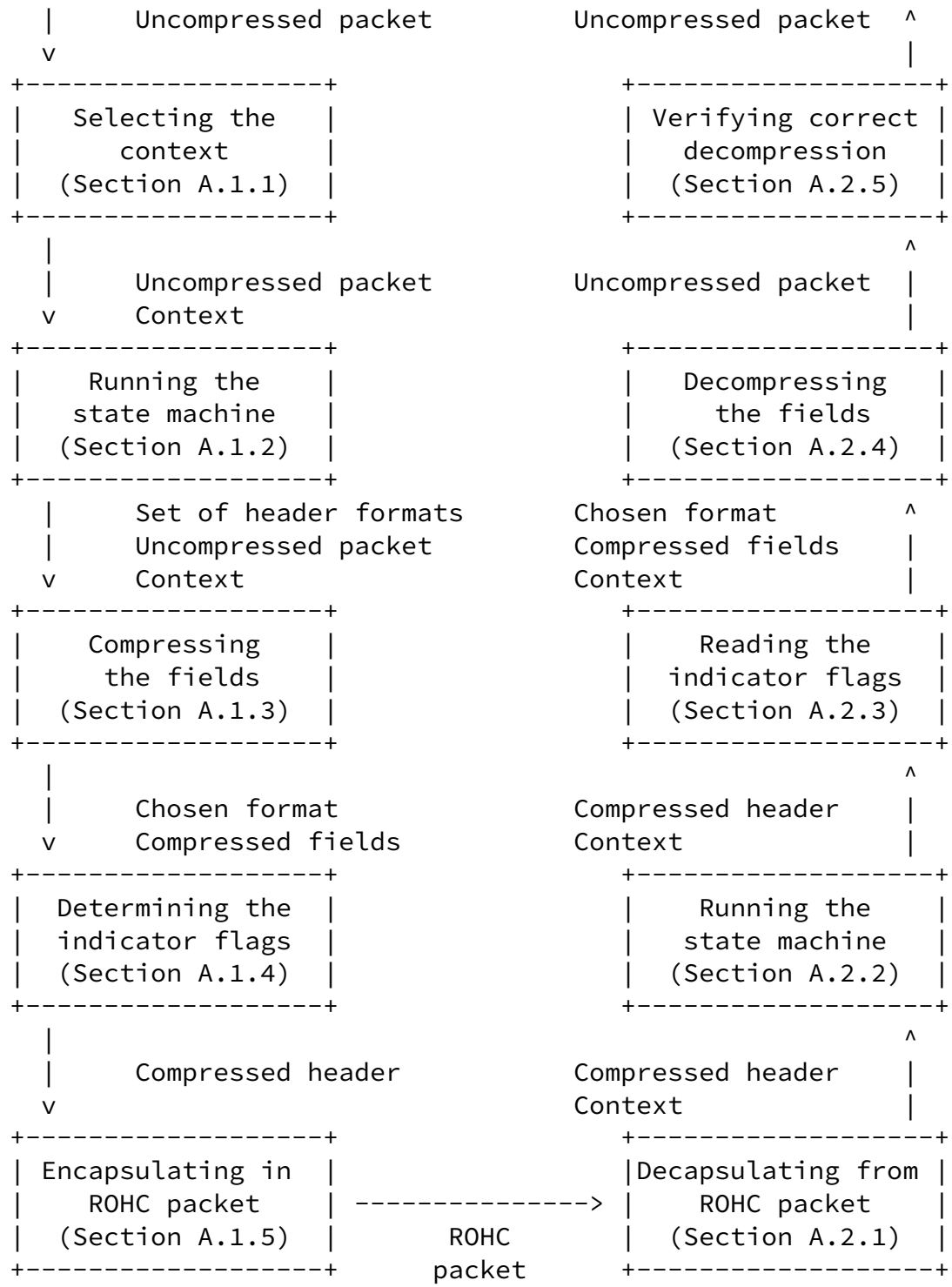

Figure 2: EPIC-LITE compression/decompression

Each of these steps is described in more detail below.

At the compressor:

Step 1: The input to the compressor is simply an uncompressed packet (with known length). In order to compress the packet it is first necessary to classify it and choose the context relative to which the packet will be compressed. If no suitable context is available then an existing context must be overwritten.

Step 2: Once the context has been chosen the compressor knows which ROHC [1] profile will be used to compress the packet. In particular it can run the state machine that determines which set of header formats will be used to compress the packet (IR, IR-DYN or CO).

Step 3: Given the uncompressed packet and a set of compressed header formats, the compressor can choose a header format to robustly carry this information to the decompressor using as few bits as possible. Note that EPIC-LITE chooses the header format simultaneously with compressing the header to improve the overall speed of compression.

Step 4: Each compressed header format has a unique set of indicator flags that communicate to the decompressor which format is in use. The compressor determines these indicator flags and appends them to the front of the compressed fields to create a compressed header.

Step 5: Once the compressed header has been calculated, the compressor encapsulates it within a ROHC packet by adding 0 or more octets of context identifier together with any padding and segmentation that is required.

At the decompressor:

Step 1: The input to the decompressor is a ROHC packet. From this packet the decompressor can determine the attached context identifier, which in turn specifies the context relative to which the packet should be decompressed.

Step 2: Once the context has been identified, the decompressor can run the state machine to determine if the packet may be decompressed.

Step 3: The decompressor then reads the indicator flags in the header to determine which compressed header format has been used. This allows the compressed value of each field to be extracted.

Step 4: Using the compressed value of each field and the context, the decompressor can apply the encoding methods to reconstruct the uncompressed packet. Note that fields are decompressed in reverse order to compression (this ensures that fields which are inferred from other field values are reconstructed correctly).

Step 5: Finally, the decompressor verifies that correct decompression

has occurred by applying the header checksum. If the packet is successfully verified then it can be forwarded.

3.3 BNF input language for creating new ROHC profiles

EPIC-LITE is a protocol-independent compression scheme because it can generate new compressed header formats automatically using a special algorithm. In order for EPIC-LITE to compress a new protocol stack however, it must be given a description of how the stack behaves.

EPIC-LITE uses a simple BNF-based input language for the fast creation of new compression schemes. The language is designed to be easy to use without detailed knowledge of the mathematics underlying EPIC-LITE. The only information required to create a new ROHC [1] profile using EPIC-LITE is a description of how the chosen protocol stack behaves.

EPIC-LITE converts the input into one or more sets of compressed header formats that can be used by a ROHC [1] compressor and decompressor. As with all version of BNF the input language has a rule-based structure, which makes it easy to build new profiles out of existing ones (e.g. when adding new layers to a protocol stack).

Figure 3 describes the process of building a set of compressed header formats from the BNF input, which is done once only and the results stored at the compressor and decompressor. References are given to pseudocode in Appendix A which describes the various stages explicitly.

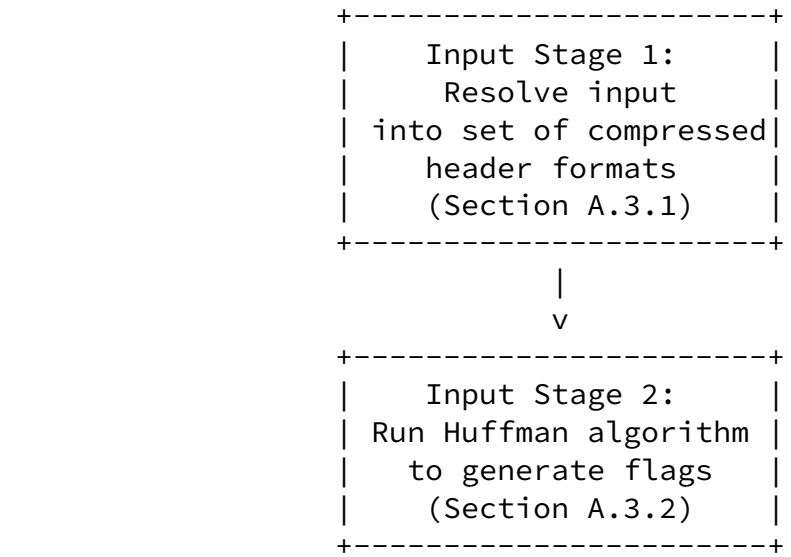

Figure 3: Building EPIC-LITE compressed header formats

Note that since the EPIC-LITE compressed header formats can be generated offline, the fact that profiles are specified using an input language does not affect the processing requirements of compression and decompression.

3.4 Huffman compression

Huffman compression [2] is a well known technique used in many popular compression schemes. EPIC-LITE uses ordinary Huffman compression to generate a new set of compressed header formats.

The basic Huffman algorithm is designed to compress an arbitrary stream of characters from a character set such as ASCII. The idea is to create a new set of compressed characters, where each normal character maps onto a compressed character and vice versa. Common characters are given shorter compressed equivalents than rarely used characters, reducing the average size of the data stream.

EPIC-LITE uses Huffman compression to generate the indicator flags for each compressed header format. Each format is treated as one character in the Huffman character set, so more common compressed header formats are indicated using fewer bits than rare header formats. The most commonly used header format is often indicated by the presence of a single "0" flag at the front of the compressed header.

The following chapters describe the mechanisms of EPIC-LITE in greater detail.

4. Overview of the BNF input language for EPIC-LITE

This chapter describes the BNF-based input language provided by EPIC-LITE for the creation of new ROHC [1] profiles.

The language is designed to be flexible but at the same time easy to use without detailed knowledge of the mathematics underlying EPIC-LITE. The only requirement for writing an efficient EPIC-LITE profile is a description of how the relevant protocol stack behaves.

As with all versions of BNF, the description of the protocol is built up using the following two constructs:

New BNF rule	A new encoding method is created from existing ones by writing a new BNF rule for the encoding method
Set of choices	One or more encoding methods can be assigned to a given field by using the choice ("|") operator

EPIC-LITE also contains a library of fundamental encoding methods (STATIC compression, LSB compression etc.) as described in Chapter 5. The BNF description of how to compress a new protocol stack always resolves into a selection of these fundamental encoding methods.

The exact syntax of the BNF-based input language can itself be described using an existing version of BNF. A suitable variant is "Augmented BNF" (ABNF) as described in RFC-2234 [6]. For example, in ABNF the syntax for defining a new encoding method in terms of existing ones is as follows:

```
<encoding_method>       =    <encoding_name> <ws> "="
                             1*(<ws> <field_encoding>)

<field_encoding>        =    <encoding_name>
                             *(<ws> "|" <ws> <encoding_name>)
```

Each instance of <encoding_name> calls an encoding method that converts a certain number of uncompressed bits into a certain number of compressed bits. Note also that <ws> is white space, used to delimit the various encoding methods.

A complete description of the BNF-based input language can be found

in Appendix A.5.

An example of how to create a new encoding method is given below. An encoding method known as IPv6_Header is constructed from the basic encoding methods available in the library. This new method is designed to compress an entire IPv6 header.

```
IPv6_Header_co       =       Version
                             Traffic_Class_co
                             ECT_Flag_co
                             CE_Flag
                             Flow_Label_co
                             Payload_Length
                             Next_Header
                             Hop_Limit_co
                             Source_Address_co
                             Destination_Address_co

Version              =       VALUE(4,6,100%)

Traffic_Class        =       STATIC(100%)

ECT_Flag             =       STATIC(100%)

CE_Flag              =       VALUE(1,0,99%) | VALUE(1,1,1%)

Flow_Label           =       STATIC(100%)

Payload_Length       =       INFERRED-SIZE(16,288)

Next_Header          =       STACK-TO-CONTROL(8)

Hop_Limit            =       STATIC(100%)

Source_Address       =       STATIC(100%)

Destination_Address  =       STATIC(100%)
```

Each field in the IPv6 header is given a choice of possible encoding methods. If an encoding method is not implicitly used 100% of the time for that field (e.g. INFERRED-SIZE) then one of the parameters is the probability that it will be used to encode the field in question. This is very important since EPIC-LITE ensures that common encoding methods require fewer bits to carry the compressed data than rarely used encoding methods.

For optimal compression, the probability should equal the percentage

of time for which the encoding method is selected to compress the

field. Note that there is no requirement for probabilities to add up to exactly 100%, as EPIC-LITE will automatically scale the probabilities by a constant factor if they do not.

Note also that the BNF input language is designed to be both human-readable and machine-readable. If only one protocol stack needs to be compressed, the input language can simply be converted by hand directly to an implementation. However, since the input language provides a complete description of the protocol stack to be compressed, it is possible to compress headers using only the information contained in the BNF description and without any additional knowledge of the protocol stack. This means that it is possible to implement a protocol-independent compressor that can download a new ROHC [1] profile described in the BNF input language and immediately use it to compress headers.

4.1 Information stored at compressor and decompressor

Any ROHC compressor maintains a number of contexts as described in Section 5.1.3 of ROHC [1]. Each context at the compressor and decompressor includes the following:

```
Compression profile: Compressed header formats
                     State machine
Field values:        One or more previously processed headers
```

The compression profile describes how to compress a certain protocol stack over a certain type of link. It includes the profile parameters that describe the set of compressed header formats (as discussed in Chapter 6) and additionally records the current state of the state machine.

The compressor also stores one or more sets of field values from previously processed headers. Each new header can be compressed relative to these field values to improve the compression ratio.

For the profiles generated using EPIC-LITE, the compressor and decompressor maintain a context value for some or all of the "field encodings" specified in the BNF description (recall that a field encoding is a set of one or more encoding methods that can be used to compress a given field). This context value is taken from the last header to be successfully compressed or decompressed.

Furthermore, in order to provide robustness the compressor can maintain more than one context value for each field. These values represent the r most likely candidates values for the context at the

decompressor, given that bit errors and dropped packets may prevent the compressor from being 100% certain exactly which values are contained in the decompressor context.

EPIC-LITE ensures that the compressed header contains enough information so that the uncompressed header can be extracted no matter which one of the compressor context values is actually stored at the decompressor. The only problem arises if the decompressor has a context value that does not belong to the set of values stored at the compressor; this situation is detected by a checksum over the uncompressed header and the packet is discarded at the decompressor.

If more than one value for a field is stored in the compressor context, some of the library encoding methods will automatically fail or only succeed under certain conditions. For example, STATIC encoding will fail and LSB encoding will only succeed if sufficient LSBs are sent to infer correct value of the field regardless of the precise value stored in the decompressor context.

Note that the rules for extracting fields from the uncompressed header and updating the context values are given in Appendix A.

The number of context values per field to be stored at the compressor is implementation-specific. Storing more values reduces the chance that the decompressor will have a context value different from any of the values stored at the compressor (which could cause the packet to be decompressed incorrectly). The trade-off is that the compressed header will be larger because it must contain enough information to decompress relative to any of the candidate context values.

As an example, an implementation may choose to store the last r values of each field in the compressor context. In this case robustness is guaranteed against up to r - 1 consecutive dropped packets between the compressor and the decompressor.

4.2 Generated data

There is some data that is passed from the compressor to the decompressor, but which is not present in the uncompressed header. This data communicates additional information that might be useful to the decompressor: for example a checksum over the uncompressed header to ensure correct decompression has occurred.

This data may be generated by certain encoding methods and then added either to the uncompressed header to be compressed immediately or to the control data stack to be compressed later.

5. Library of EPIC-LITE encoding methods

The ROHC [1] standard contains a number of different encoding methods (LSB encoding, scaled timestamp encoding, list-based compression etc.) for compressing header fields. EPIC-LITE treats these encoding methods as library functions to be called by the BNF input language when they are needed.

The following library contains a wide range of basic encoding methods. Moreover new encoding methods can be added to the library as and when they are needed.

Note that this chapter contains an informative description only. The normative pseudocode description of every encoding method can be found in Appendix A.4.

The syntax of each encoding method is given using ABNF as defined in RFC-2234 [6]. Note that each of the encoding methods may have one or more parameters of the following type:

<value>	A non-negative integer (specified as decimal, binary or hex). Binary values are prefixed by 0b and hex values are preceded by 0x.
<offset>	An integer (positive or negative)
<length>	A non-negative integer used to indicate the length of the field being compressed
<probability>	A probability expressed as a percentage with at most 2 decimal places
<encoding_name>	The name of another encoding method including all parameters

The ABNF description of these parameters is found in Appendix A.5.

5.1 STATIC

ABNF notation: "STATIC(" <probability> ")"

The STATIC encoding method can be used when the header field does not change relative to the context. If a field is STATIC then no information concerning the field need be transmitted in the compressed header.

The only parameter for the STATIC encoding method is a probability that indicates how often the method will be used. Encoding methods with high probability values require fewer bits in the compressed header than encoding methods that are allocated low probability values. In general the probability should reflect as accurately as possible the chance that the field will be encoded as STATIC.

5.2 IRREGULAR

ABNF notation: "IRREGULAR(" <length> "," <probability> ")"

The IRREGULAR encoding method is used when the field cannot be compressed relative to the context, and hence must be transmitted in full in the compressed header.

The IRREGULAR encoding method has a length parameter to indicate the length of the field in bits and a probability that indicates how often it will be used.

A modified version of IRREGULAR encoding is given below:

5.2.1 IRREGULAR-PADDED

ABNF notation: "IRREGULAR-PADDED(" <length> "," <value> "," <probability> ")"

The IRREGULAR-PADDED encoding method compresses any field that is large in terms of number of bits but has a small actual value (and hence the most significant bits are zero).

The encoding method transmits a certain number of LSBs (Least Significant Bits) of the field. The first parameter gives the overall length of the field, whilst the next parameter specifies the number of LSBs to be transmitted in the compressed header. The bits not transmitted are all taken to be 0 by the decompressor. The probability gives an indication of how often IRREGULAR-PADDED will be used.

The IRREGULAR-PADDED encoding method is useful for compressing fields that take small integer values with a high probability.

5.3 VALUE

ABNF notation: "VALUE(" <length> "," <value> "," <probability> ")"

The VALUE encoding method can be used to transmit one particular value for a field. It is followed by parameters to indicate the length and integer value of the field as well as the probability of the field taking this value.

5.4 LSB

ABNF notation: "LSB(" <value> "," <offset> "," <probability> ")"

The LSB encoding method compresses the field by transmitting only its LSBs (Least Significant Bits).

The first parameter indicates the number of LSBs to transmit in the compressed header. The second parameter is the offset of the LSBs: it describes whether the decompressor should interpret the LSBs as increasing or decreasing the field value contained in its context. Again the probability indicates how often LSB encoding will be used.

To illustrate how the second parameter works, suppose that k LSBs are transmitted with offset p. The decompressor uses these LSBs to replace the k LSBs of the value of this field stored in the context (val), and then adds or subtracts multiples of 2^k so that the new field value lies between (val - p) and (val - p + 2^k - 1).

In particular, if p = 0 then the field value can only stay the same or increase. If p = -1 then it can only increase, whereas if p = 2^k then it can only decrease.

Recall that for robustness the compressor can store r values for each field in its context. If this is the case then enough LSBs are transmitted so that the decompressor can reconstruct the correct field value, no matter which of the r values it has stored in its context. This is equivalent to Window-based LSB encoding as described in ROHC [1].

5.5 UNCOMPRESSED

ABNF notation: "UNCOMPRESSED(" <value> "," <value> "," <value> "," <value> ")"

The UNCOMPRESSED encoding method transmits a field uncompressed without alteration. All uncompressed fields are transmitted as-is at the end of the compressed header.

The UNCOMPRESSED encoding method differs from the IRREGULAR encoding method in that the size of the field is not fixed, but instead is

specified by a control field. The first parameter gives the length n of the control field: UNCOMPRESSED encoding obtains this control field simply by removing the first value (which should be n bits) from the control data stack.

The next three parameters specify a divisor, multiplier and offset for the control field. These parameters scale the value of the control field so that it specifies the exact size of the UNCOMPRESSED field in bits. If the parameters are d, m and p respectively then:

size of UNCOMPRESSED field = floor(control field value / d) * m + p

UNCOMPRESSED encoding is usually used in conjunction with one of the STACK encoding methods, which write to the control data stack as explained below:

5.6 STACK encoding methods

These methods are used to move values around for use by future encoding methods. They take as a parameter the number of bits to be transferred and always have 100% probability of being used.

5.6.1 STACK-TO-CONTROL

ABNF notation: "STACK-TO-CONTROL(" <length> ")"

This encoding method takes the specified number of bits from the uncompressed header and transfers them to the control data stack. It does the reverse at the decompressor.

5.6.2 STACK-FROM-CONTROL

ABNF notation: "STACK-FROM-CONTROL(" <length> ")"

This encoding method takes an item with the specified number of bits from the control data stack and transfers it to the uncompressed header. It does the reverse at the decompressor.

5.6.3 STACK-PUSH-MSN

ABNF notation: "STACK-PUSH-MSN(" <length> ")"

The MSN (Master Sequence Number) is a width defined value that increases by one for each packet received at the compressor.

This encoding method copies the least significant specified number of bits of the MSN to the top of the control data stack. Conversely, it removes these bits at the decompressor.

5.6.4 STACK-POP-MSN

ABNF notation: "STACK-POP-MSN(" <length> ")"

This encoding method removes the specified number of bits of the MSN from the uncompressed data or adds it at the decompressor.

5.6.5 STACK-ROTATE

ABNF notation: "STACK-ROTATE(" <value> "," <value> ")"

This encoding method rotates the top n items on the top control stack m times where n is the first parameter and m the second. A rotation of n items by 1 moves the nth element on the control stack to the top of the stack (and the top n-1 items down one place).

5.7 INFERRED encoding methods

The following versions of INFERRED encoding are available:

5.7.1 INFERRED-TRANSLATE

ABNF notation: "INFERRED-TRANSLATE(" <length> "," <length> *("," <value> "," <value>) ")"

The INFERRED-TRANSLATE encoding method translates a field value under a certain mapping. The first pair of parameters specifies the length of the field before and after the translation. This is followed by additional pairs of integers, representing the field value before and after it is translated. Note that the final field value at the compressor (or equivalently, the original field value at the decompressor) appears first in each pair. For example:

```
INFERRED-TRANSLATE(8,16,41,0x86DD,4,0x0800)     ; GRE Protocol
```

The GRE Protocol field behaves in the same manner as the Next Header field in other extension headers, except that it indicates that the subsequent header is IPv6 or IPv4 using the values 0x86DD and 0x0800 instead of 41 and 4. The INFERRED-TRANSLATE encoding method can convert the standard values (as provided by LIST compression defined

in Section 5.11) into the values required by the GRE Protocol field.

At the compressor, once the translation is complete the field is copied to the control data stack. At the decompressor the field is

removed from the control data stack, translated and then added to the uncompressed data.

5.7.2 INFERRED-SIZE

ABNF notation: "INFERRED-SIZE(" <length> "," <offset> ")"

The INFERRED-SIZE encoding method infers the value of a field from the size of the uncompressed packet.

The first parameter specifies the uncompressed field length in bits, and the second parameter specifies the offset of the uncompressed packet size (i.e. the amount of packet which has already been compressed). If the INFERRED-SIZE field value is v, the offset is p and the total packet length after (but not including) the INFERRED-SIZE field is L then the following equation applies (assuming 8 bits in a byte):

L = 8 * v + p

5.7.3 INFERRED-OFFSET

ABNF notation: "INFERRED-OFFSET(" <length> ")"

The INFERRED-OFFSET encoding method compresses a field that usually has a constant offset relative to a certain base field.

The parameter describes the length of the field to be compressed. The base field will already be on the control data stack - put there using one of the STACK methods.

The encoding subtracts the base field from the field to be compressed and replaces the field value by these "offset" bits in the uncompressed header. The offset bits are then compressed by the next encoding method in the input code.

For example, a typical sequence number can be compressed as follows:

```
            STACK-PUSH-MSN(32)                ; Put MSN on control stack

            INFERRED-OFFSET(32)               ; Sequence Number

            STACK-POP-MSN(32)                 ; Remove MSN

            STATIC(99%) |                     ; Sequence Number offset
            LSB(8,-1,0.7%) |
            LSB(16,-1,0.2%) |
            IRREGULAR(32,0.1%)
```

In this case the offset field is expected to change rarely and only by small amounts, and hence it is compressed using mainly STATIC and LSB encodings.

5.7.4 INFERRED-IP-CHECKSUM

ABNF notation: "INFERRED-IP-CHECKSUM(" <encoding_name> ")"

This method is used for calculating the IP checksum. At the compressor it replaces the bits representing the IP checksum with "0" bits (these are a known distance from the beginning of this method). It then continues to compress using encoding_name. At the decompressor it decompresses encoding_name as usual. A 16-bit one's complement sum is then computed over the length of data which has been decompressed in this method and the result copied into the appropiate bits (at a fixed offset) in the header.

5.8 NBO

ABNF notation: "NBO(" <length> ")"

The NBO encoding method may be used if there is a possibility that the following field will be in reverse byte order from the rest of the header, as is sometimes seen with the IP ID, for example. The method looks at the specified number of bits (typically 16 or 32) and, using a history, decides whether or not they are in network byte order. If they are it removes them from the stack, puts a 1-bit flag with the value "1" on the stack and replaces the original value. If the value is not in NBO it removes the bits, puts a flag with the value "0" on the stack and replaces the byte-swapped value. This method can be used prior to other methods, for example:

```
            NBO (16)                        ; Check for network byte order

            STACK-PUSH-MSN(16)              ; Put MSN on control stack

            INFERRED-OFFSET(16)             ; IP ID

            STACK-POP-MSN(16)               ; Remove MSN

            STATIC(80%) |                   ; IP ID offset
            LSB(5,-1,15%) |
            IRREGULAR(16,5%)

            STATIC(99%) |                   ; NBO flag
            IRREGULAR(1,1%)
```

5.9 SCALE

ABNF notation: "SCALE(" <length> ")"

The SCALE encoding method is used for fields which change by a regular (usually large) amount every packet. The number of bits specified are removed and a suitable scale factor chosen. Three values each with the specified length are then replaced on the stack. If the original field has value v and the chosen scale is s then these values are:

v / s = the scaled value of v when v is divided by s using integer arithmetic

v mod s = the remainder when v is divided by s

s = the scale value

They are placed on the stack in the order above so that the next value to be compressed is the scale value.

5.10 OPTIONAL

ABNF notation: "OPTIONAL(" <encoding_name> ")"

The OPTIONAL encoding method is used to compress fields that are optionally present in the uncompressed header.

OPTIONAL encoding requires a 1 bit indicator flag to specify whether or not the optional field is present in the uncompressed header.

This flag is extracted from the control data stack. The value of the flag is added to the stack by another encoding method (such as STACK-TO-CONTROL or LIST). For example:

```
STACK-TO-CONTROL(1)              ; GRE Key flag

OPTIONAL(KEY-ENCODING)           ; GRE Key
```

In this case the encoding method KEY-ENCODING is called to compress the GRE Key field, but only if the Key Flag is set to 1. If the Key Flag is set to 0 (indicating that the GRE Key is not present) then some padding bits of 0 may be added to the compressed header (the number of bits added is determined by the compressor - the compressor chooses a format for KEY-ENCODING, which though not used provides the number of bits with which to pad). If there is a U method encompassing the OPTIONAL method then the number of bits with which to pad is automatically zero.

5.11 MANDATORY

ABNF notation: "MANDATORY(" <encoding_name> ")"

This encoding method may be used where another encoding method has appended a flag indicating the presence of a field in the uncompressed header. If the field is optionally present then the OPTIONAL encoding (above) may be used. If the field is always present then the MANDATORY encoding can be used. This checks that the value of the flag on the stack is 1 (indicating that the field is present). If the value of the flag is 0 then the MANDATORY encoding method will fail.

5.12 CONTEXT

ABNF notation: "CONTEXT(" <encoding_name> "," <value> ")"

The CONTEXT encoding method is used to store multiple copies of the same field in the context. This encoding method is useful when compressing fields that take a small number of values with high probability, but when these values are not known a-priori.

CONTEXT encoding can also be applied to larger fields: even an entire TCP header. This can be very useful when multiple TCP flows are sent to the same IP address, as a single ROHC [1] context can be used to compress the packets in all of the TCP flows.

The first parameter specifies the encoding method that should be used to compress the field. The second parameter specifies how many copies of the field should be stored in the context.

CONTEXT encoding applies the specified encoding method to the uncompressed header, compressing relative to any of the copies of the field stored in its context. It then appends an "index" value to the uncompressed header to indicate to the decompressor which context value should be used for decompression. Consider the following example using the TCP Window field:

```
CONTEXT(TCP-WINDOW,4)              ; Window

VALUE(2,0,89%) |                   ; Window context index
VALUE(2,1,10%) |
VALUE(2,2,0.5%) |
VALUE(2,3,0.5%)
```

At most 4 copies of the Window field can be stored in the context. The Window field can be compressed relative to any of these values: the value chosen by the compressor is transmitted to the decompressor using the "index".

5.13 LIST

ABNF notation: "LIST(" <value> "," <value> "," <value> "," <value> "," <value> "," <encoding_name> *("," <encoding_name>) *("," <value>) ")"

The LIST encoding method compresses a list of items that do not necessarily occur in the same order for every uncompressed header. Example applications for the LIST encoding method include TCP options and TCP SACK blocks.

The size of the list is determined by a control field in exactly the same manner as for UNCOMPRESSED encoding. The first four integer parameters are defined as in UNCOMPRESSED.

The fifth parameter gives the number of bits which should be read from the uncompressed_data stack to decide which of the encoding methods in the list to use to compress the next list item.

These parameters are followed by a set of encoding methods that can be used to compress individual items in the list and a set of integers to identify which method to use. If the integer obtained using the fifth parameter matches the nth integer then use the nth

encoding method.

This continues until data amounting to the size of the list has been compressed.

Once the list size is reached, LIST encoding appends the order in which the encoding methods were applied to the uncompressed data and the presence of data for the methods. The profile defines whether the order and presence can change in a compressed packet or not. For example:

```
LIST(4,1,32,0,8,
     OPTIONAL(TCP-SACK),
     OPTIONAL(TCP-TIMESTAMP),
     OPTIONAL(TCP-END),
     OPTIONAL(TCP-GENERIC),
     5,8,0)                        ; TCP Options
```

```
            STATIC(99.9%) |                     ; TCP Options order
            IRREGULAR(8, 0.1%)

            STATIC(75%) |                       ; TCP Options presence
            IRREGULAR(4,25%)
```

The order in which the methods are applied is irrelevant in the mapping between methods chosen and the indicator flags. They are always considered to be compressed in the order in which they appear in the profile and the order and presence fields deal with the actual order in which they were processed.

5.13.1 LIST-NEXT

ABNF notation: "LIST-NEXT(" <value> "," <encoding_name> *("," <encoding_name>) *("," <value>) ")"

LIST-NEXT encoding is similar to basic LIST encoding, except that the next list item to compress is known a-priori from a control field. IP extension headers can be compressed using LIST-NEXT.

The first parameter specifies the number of bits to extract from the control data stack before each list item is compressed. This is followed by the set of encoding methods available to LIST-NEXT and a set of 0 or more integers. The nth encoding method can only be called when the nth integer value is obtained from the control data stack.

For example:

```
            LIST-NEXT(8, OPTIONAL(AH-ENCODING),
                      OPTIONAL(ESP-ENCODING),
                      OPTIONAL(GRE-ENCODING),
                      OPTIONAL(GENERIC-ENCODING),
                      51,50,47)                 ; Header Chain

            STATIC(99.9%) |                     ; Header Chain order
            IRREGULAR(8,0.01%)
```

```
            STATIC(75%) |                       ; Header Chain presence
            IRREGULAR(4,25%)
```

The IP extension header chain can have a number of specific encoding methods designed for one type of extension header (AH, ESP or GRE) as well as a "generic" encoding method that can cope with arbitrary extension headers but at reduced compression efficiency.

Just as with basic LIST encoding, LIST-NEXT also adds the order in which the encoding methods are applied and their presence or absence to the uncompressed header, so that the decompressor can reconstruct the list in the correct order.

5.14 FLAG encoding methods

The flag encoding methods are used to modify the behavior of another encoding method. Each flag encoding has a single parameter, which is the name of another encoding method. The flag encoding method calls this encoding method, but additionally modifies the input or output in some manner.

Note that flag encoding methods do not require the original encoding method to be rewritten (as they only modify its input or output).

5.14.1 N flag

ABNF notation: "N(" <encoding_name> ")"

The N flag runs the encoding method specified by its parameter, with the exception that it does not update the context. This is useful when a field takes an unexpected value for one header and then reverts back to its original behavior in subsequent headers.

An example of the N flag in use is given below:


```
            STATIC(99%) |                       ; Window
            N(LSB(11,2048,0.9%)) |
            IRREGULAR(16,0.1%)
```

In the above example the N flag is applied to the TCP Window field.

The field is compressed by transmitting only the last few LSBs, which are always interpreted at the decompressor as a decrease in the field value. However, because the context is not updated the field reverts back to its original value following the decrease.

5.14.2 U flag

ABNF notation: "U(" <encoding_name> ")"

The U flag changes the destination of any compressed bits produced by the encoding method specified as its parameter. Instead of putting them on the compressed_data stack it puts them on the unc_fields stack. They no longer contribute to the length of the compressed header so this is taken account of in the build method. This means that if an OPTIONAL method is surrounded by a U flag and the optional part is not actually present then no padding bits are required.

An example of the U flag in use is given below:

```
U(OPTIONAL(crsc_entry))          ; CSRC-list entry

where

csrc_entry = IRREGULAR(32)
```

This means that if the CSRC entry is present then it will be compressed as IRREGULAR (i.e. the full 32 bits will be sent) and these bits will be pushed on the uncompressed stack. However, if it is not present then no padding bits will need to be sent.

5.15 FORMAT

ABNF notation: "FORMAT(" <encoding_name> "," *(<encoding_name>) ")"

The FORMAT encoding method is used to create more than one set of compressed header formats.

Recall that each set of compressed header formats uses up all of the indicator bit patterns available at the start of the compressed

header. Thus a profile can have several sets of compressed header formats, but only one set can be in use at a given time.

FORMAT encoding is followed by a list of k encoding methods. Each encoding method is given its own set of compressed header formats in the CO packets. Note however that all encoding methods are present in the IR(-DYN) packets, so an IR(-DYN) packet may be sent to change to a new set of compressed header formats.

An index flag is appended to the uncompressed header to indicate which set of formats is currently in use, as illustrated by the following example of an IR(-DYN) format (for CO the index flag would be STATIC(100%)):

```
FORMAT(SEQUENTIAL-IP-ID,RANDOM-IP-ID)          ; IP ID

IRREGULAR(1,100%)                              ; IP_ID Index
```

Two sets of compressed header formats are provided: one for an IP ID that increases sequentially, and one for a randomly behaving IP ID. Note that the Index flag is only sent in the IR(-DYN) packets.

5.16 CRC

ABNF notation: "CRC(" <value> "," <probability> ")"

The CRC encoding method generates a CRC checksum calculated across the entire uncompressed header. At the decompressor this CRC is used to validate that correct decompression has occurred.

Note that it is possible for different header formats to have different amounts of CRC protection, so extra CRC bits can be allocated to protect important context-updating information. This is illustrated in the example below:

```
CRC(3,99%) |                       ; Checksum Coverage
CRC(7,1%)
```

The uncompressed header is recorded in the crc_static and crc_dynamic variables. Note that the fields that either never change or only change in the IR packet are placed in crc_static, and the remaining fields are placed in crc_dynamic. The CRC is calculated over crc_static + crc_dynamic, with the static fields placed first to speed up computation.

In general an EPIC-LITE profile can use any CRC length for which a CRC polynomial has been explicitly defined. The following CRC lengths are currently supported:

```
3-bit:      C(x) = 1 + x + x^3
6-bit:      C(x) = 1 + x + x^3 + x^4 + x^6
7-bit:      C(x) = 1 + x + x^2 + x^3 + x^6 + x^7
8-bit:      C(x) = 1 + x + x^2 + x^8
10-bit:     C(x) = 1 + x + x^4 + x^5 + x^9 + x^10
12-bit:     C(x) = 1 + x + x^2 + x^3 + x^11 + x^12
16-bit:     C(x) = 1 + x^2 + x^15 + x^16
```

5.17 MSN encoding methods

These methods are used to send the MSN (Master Sequence Number) to the decompressor. Separate encoding methods are required because the MNS does not appear in the uncompressed data itself.

There are two encoding methods which compress the MSN and one which can be used to assign a particular field to be the MSN (for example, RTP sequence number):

5.17.1 MSN-LSB

ABNF notation: "MSN-LSB(" <value> "," <offset> "," <probability> ")"

This method uses ordinary LSB encoding on the value in MSN rather than on the uncompressed_data stack. It also stores some information for use if extra bits of MSN are to be sent to pad the compressed header to a specific bit alignment.

5.17.2 MSN-IRREGULAR

ABNF notation: "MSN-IRREGULAR(" <length> "," <probability> ")"

As with MSN-LSB this method uses ordinary IRREGULAR encoding and stores the extra information described above.

5.17.3 SET-MSN

ABNF notation: "SET-MSN(" <length> ")"

The parameter of this method specifies the number of bits to assign

to be MSN. The bits are taken from the uncompressed stack. This value can then be used for INFERRED-OFFSET methods and compressed using MSN-LSB or MSN-IRREGULAR when no more fields need it.

6. Creating a new ROHC profile

This chapter describes how to generate new ROHC [1] profiles using EPIC-LITE. It is important that the profiles are specified in an unambiguous manner so that any compressor and decompressor using the profiles will be able to interoperate.

The following eight variables are required by EPIC-LITE to create a new ROHC [1] profile:

```
profile_identifier
max_formats
max_sets
bit_alignment
npatterns
CO_packet
IR_DYN_packet
IR_packet
```

Once a value has been assigned to each variable the profile is well-defined. A compressor and decompressor using the same values for each variable should be able to successfully interoperate.

Each of the variables is described in more detail below:

6.1 Profile identifier

The profile_identifier is a 16-bit integer that is used when negotiating a common set of profiles between the compressor and decompressor. Official profile identifiers are assigned by IANA to ensure that two distinct profiles do not receive the same profile

identifier. Note that the 8 MSBs of the profile identifier are used to specify the version of the profile (so that old profiles can be obsoleted by new profiles).

6.2 Maximum number of header formats

The max_formats parameter controls the number of compressed header formats to be stored at the compressor and decompressor.

If more compressed header formats are generated than can be stored then EPIC-LITE discards all but the max_formats most probable formats to be used. Note that the max_formats parameter affects the EPIC-LITE compressed header formats, and so for interoperability it MUST

be specified as part of the profile.

If more than max_formats header formats are generated for the IR packet then this means that there are headers which it may be impossible to compress using this profile (resulting in a switch to a different profile). Hence it is RECOMMENDED that no more than max_formats distinct header formats are generated for the IR packet of any profile.

In a similar manner the max_sets parameter controls the total number of sets of compressed header formats to be stored. Recall that a profile can have several sets of compressed header formats, but only one set may be in use at a given time. It is important to note that the maximum size specified by max_formats applies to each individual set of header formats, so the total overall number of formats that need to be stored is equal to max_formats * (max_sets + 2) including the 2 sets of formats for the IR and IR-DYN packets.

6.3 Control of header alignment

The alignment of the compressed headers is controlled using the bit_alignment parameter. The output of the EPIC-LITE compressor is guaranteed to be an integer multiple of bit_alignment bits long.

Additionally, the parameter npatterns can be used to reserve bit patterns in the compressed header. The parameter specifies the number of bit patterns in the first word (i.e. the first bit_alignment bits) of the compressed header that are available for

use by EPIC-LITE. Consequently npatterns takes a value between 1 and 2^bit_alignment.

For compatibility with ROHC [1], it is important for EPIC-LITE not to use the bit patterns 111XXXXX in the first octet of each compressed header because they are reserved by the ROHC framework. So to produce a set of header formats compatible with ROHC [1] the bit_alignment parameter MUST be set to 8 and npatterns MUST be set to 224.

6.4 Compressed header formats

The profile parameter CO_packet specifies an encoding method that is used to generate the EPIC-LITE CO packets. This encoding method may be described using the BNF-based input language provided in Chapter 4 (or in fact can be described in any manner provided that it is unambiguous).

The distinction between the eight variables required to define a new ROHC [1] profile and the input language defined in Chapter 4 is

important. The only requirement for compatibility with the profile is that the correct compressed header formats are used: the fact that they are specified in the input language is not important, and they can be implemented in any manner.

The profile parameters IR_packet and IR_DYN_packet specify an encoding method which is used to generate the EPIC-LITE IR and IR-DYN packets respectively.

Note that the IR_DYN_packet parameter is optional. If it is not given then EPIC-LITE generates the IR-DYN packet using the same encoding method as specified by the CO_packet parameter. The IR_packet parameter is also optional. If it is not given then EPIC-LITE generates the IR packet using the same encoding method as specified by the IR_DYN_packet parameter (or CO_packet if IR_DYN_packet is also not given).

7. Security considerations

EPIC-LITE generates compressed header formats for direct use in ROHC profiles. Consequently the security considerations for EPIC-LITE

inherit those of ROHC [1].

EPIC-LITE profiles also describe how to compress and decompress headers. As such they are interpreted or compiled by the compressor and decompressor. An error in the profile description may cause undefined behaviour. In a situation where profiles could be dynamically updated consideration MUST be given to authenticating and verifying the integrity of the profile.

8. Acknowledgements

Header compression schemes from ROHC [1] have been important sources of ideas and knowledge. Basic Huffman encoding [2] was enhanced for the specific tasks of robust, efficient header compression.

Thanks to

Carsten Bormann (cabo@tzi.org)
Christian Schmidt (christian.schmidt@icn.siemens.de)
Max Riegel (maximilian.riegel@icn.siemens.de)

for valuable input and review.

References

[1] Bormann, C., Burmeister, C., Degermark, M., Fukushima, H., Hannu, H., Jonsson, L-E., Hakenberg, R., Koren, T., Le, K., Liu, Z., Martensson, A., Miyazaki, A., Svanbro, K., Wiebke, T., Yoshimura, T. and H. Zheng, "RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed", RFC 3095, July 2001.

[2] Nelson, M. and J-L. Gailly, "The Data Compression Book", M&T Books , 1995.

[3] Jacobson, V., "Compressing TCP/IP headers for low-speed serial links", RFC 1144, February 1990.

[4] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.

[5] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[6] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

Authors' Addresses

Richard Price
Siemens/Roke Manor
Roke Manor Research Ltd.
Romsey, Hants SO51 0ZN
UK

Phone: +44 (0)1794 833681
EMail: richard.price@roke.co.uk
URI: http://www.roke.co.uk

Robert Hancock
Siemens/Roke Manor
Roke Manor Research Ltd.
Romsey, Hants SO51 0ZN
UK

Phone: +44 (0)1794 833601
EMail: robert.hancock@roke.co.uk
URI: http://www.roke.co.uk

Stephen McCann
Siemens/Roke Manor
Roke Manor Research Ltd.
Romsey, Hants SO51 0ZN
UK

Phone: +44 (0)1794 833341

EMail: stephen.mccann@roke.co.uk
URI: http://www.roke.co.uk

Abigail Surtees
Siemens/Roke Manor
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

Paul Ollis
Siemens/Roke Manor
Roke Manor Research Ltd.
Romsey, Hants SO51 0ZN
UK

Phone: +44 (0)1794 833168
EMail: paul.ollis@roke.co.uk
URI: http://www.roke.co.uk

Mark A. West
Siemens/Roke Manor
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

Appendix A. EPIC-LITE compressor and decompressor

This appendix gives a complete pseudocode description of the EPIC-LITE compressor and decompressor.

The appendix contains the following sections:

   Compressor operation (Section A.1)
   Decompressor operation (Section A.2)
   Offline processing (Section A.3)
   Library of methods (Section A.4)
   BNF description of input language (Section A.5)

Recall that each EPIC-LITE profile for ROHC [1] is described by the following eight variables:

profile_identifier	16-bit integer uniquely identifying the ROHC profile generated by EPIC-LITE
max_formats	Maximum number of header formats per set
max_sets	Total number of sets of header formats
bit_alignment	Number of bits for alignment (all compressed headers will be multiples of bit_alignment bits). (Set to 8 for compatibility with ROHC [1].)
npatterns	Number of bit patterns available for EPIC-LITE in the first word of the compressed header (set to 224 for compatibility with ROHC [1].)
CO_packet	Name of the method that generates the CO packet formats
IR_DYN_packet	Name of the method that generates the IR-DYN packet formats
IR_packet	Name of the method that generates the IR packet formats

Additionally, the following general functions are used in the pseudocode description of EPIC-LITE:

The functions used for list manipulation are given below:

empty (list)	Empties the list
sort-natural (list, compare)	Sorts the list as defined by the compare function (which compares 2 elements). Sort is 'natural' in that original ordering of elements is preserved in the event of 2 elements being equal
append (list, item)	Appends an item to a list
prepend (list, item)	Prepend an item to head of list
concatenate (list, list)	Appends all the entries from the second list to the first one
copy (list, list)	Replaces the first list with a copy of the second list
head-of (list)	Finds first item in list
tail-of (list)	Finds last item in list
next-item (curr_item)	Finds the next item in the list from curr_item
previous-item (curr_item)	Finds the previous item in the list from curr_item
at-end (list)	Checks whether at the end of the list
size-of (list)	Returns the number of elements in list

The following functions are used to traverse the BNF input language describing the new ROHC profile. Note that the relevant information can be extracted from the input language by hand or automatically via a suitable BNF parser:

first-field (method_name)
finds the first instance of <field_encoding> referenced in the encoding method (applies to a non-library encoding method only)

next-field (method_name)
finds the next instance of <field_encoding> in the method (if none can be found then NULL-ENCODING is returned)

prev-field (method_name)
finds the previous instance of <field_encoding> in the method

last-field (method_name)
finds the last instance of <field_encoding> in the method (these functions allow iteration over the different field encodings in a method. This must be in the order defined in the profile)

first-method (field_encoding)
find the first encoding method listed within the field encoding

next-method (field_encoding)
find the next encoding method listed within the field encoding (this and the previous function allow iteration over the encoding methods for a given field. This must be in the order defined in the profile. If none can be found then NULL-METHOD is returned)

extract-name (method)
finds the name of the encoding method

extract-probability (method)
finds the probability of the method being invoked

Method function handling:

lookup-build-function (method)
: finds the 'BUILD' function that relates to the method. If this is a library method, then the function returns a reference to the 'BUILD' subfunction for that method. Otherwise it returns a reference to the main 'BUILD' function which is then called recursively

lookup-compress-function (method)
: as above but for the 'COMPRESS' function

lookup-decompress-function (method)
: as above but for the 'DECOMPRESS' function

context-size (method)
: looks up the number of context entries that will be needed to compress using this method. This can be worked out off line as part of the 'BUILD' function and stored for reference during compression and decompression.

count-bits (method, enc)
: counts the number of bits that would be present in a compressed header for data compressed with format enc for this method (This can be worked out off line as part of the 'BUILD' function and stored for use during compression and decompression).

Data handling:

Value based functions:

length (var)	Returns the length in bits of var
value (var)	Returns the value of var
str (len, val)	Returns a variable with length len and containing the value val
lsb (var, k)	Returns the least significant k bits of a (width defined or not) value (var) in a width defined value
msb (var, k)	Returns the most significant k bits of a width defined value (var) in a width defined value
concat (var_1, var_2)	Returns a width defined value with most significant bits being var_1 and least being var_2 - ie concatenates the two strings

Addition, subtraction and multiplication can be done on width defined values as long as the two values have the same length (i.e. this

translates into modular arithmetic) but must be done carefully.

Stack functions:

In the functions below an item has a length and value associated with it which can be found using the functions above.

push (stack_name, n, var)	For a bit-overlay stack, this function adds n bits with value var to the top of the stack (stack_name). For an item stack, this function pushes an item with length n and value var to the top of the stack (stack_name)
push (stack_name, item)	For a bit-overlay stack, this function adds n bits with value var to the top of the stack (stack_name) where n is the length of item and it has value var. For an item stack, this function pushes item on var to the top of the stack (stack_name)
pop (stack_name, n, item)	Pop n bits of data off a bit based stack (stack_name) and put into item

pop (stack_name, item)	Pop item off an item based stack (stack_name)
top (stack_name, n)	Returns the item made up of the top n bits on a bit based stack (stack_name)
top (stack_name)	Returns the top item from an item based stack (stack_name)
stack-size (stack)	Returns the size in bits of a bit based stack
add (stack_1, stack_2)	Pushes stack_1 onto stack_2 but leaves the outcome in stack_1

rotate (stack_name, n, m)	Rotate the top n items on an item based stack (stack_name) m times. Take the nth item from the stack and put it on the top.
stack-pointer (stack_name)	Returns the position of the top of a stack (stack_name) NB this is never used for accessing the stack and can be of arbitrary form

Context based functions:

The enc_index counter is used to access information stored in the context for each field encoding. It is initially 1 and is incremented throughout compression and decompression. Some encoding methods do not have any context associated with them, for example, INFERRED-SIZE. This incremental way of accessing context information means that if a choice of two or more encoding methods is available for a given field, each in turn should contain the same number of subfields as each other, otherwise the alignment of the context will be lost.
NB leaving a value in the context empty is NOT the same as storing something which zero-bits wide.

context (enc_index, i, var)	Put the value stored at the ith position in the context associated with enc_index into var. If there isn't a value in the ith position then return empty
save (enc_index, var, storage)	Store the value in var in storage

	associated with enc_index (to be transferred to the context if (de)compression is successful
clear (enc_index, storage)	Remove the value in storage associated with enc_index and leave it empty (for use if non-context-updating method)
transfer (storage, context)	Transfer the information in storage into the context

Miscellaneous:

convert-percentage (pc)	Converts the percentage (in floating point) to the fixed point representation used
store (list, method, field_encoding)	Add method associated with field_encoding to list
get-method (list, field_encoding)	Return the method associated with field_encoding from list
get-method-list (main_list_elt, method_chosen)	Finds method_chosen such that method_chosen <=> main_list_item.id
format-get-method-list (main_list_elt, method_chosen, method)	Finds method_chosen such that method_chosen <=> main_list_item.id assuming that a specific method has been chosen using the FORMAT method (this is used to ensure correct decompression of an IR(-DYN) packet)

get-format (main_list_elt, method_chosen)

Finds the main_list_item such that method_chosen <=> main_list_item.id

get-header-length (method_chosen)
Returns the length in bits of the compressed header format (excluding flags) for this list element (sum of number of bits in each compressed field)

lookup-crc-function (n)
Finds the function for computing an n-bit crc over a specified amount of data and putting the value into a given width defined variable

byte-swap (item)
Returns another item the same as item but stored in the opposite byte ordering

compute-16-checksum (data, length)
Compute a 16-bit one's complement sum over data of length

Other constructs:

```
foreach item in [reverse] list
             :
end loop
```

provides for iteration over all the elements of a list [in reverse order]

call function (...)

indicates a reference to a function is being invoked

choose (...)

the choice of whatever is specified is implementation specific - it may affect efficiency but whatever choice is made will not affect interoperability, for example, choose (j < k) - pick an arbitrary value j which is less than k.

Some global variables:

uncompressed_data	Bit based stack Compression - initially header and payload, finally empty Decompression - initially empty, finally original header and payload
compressed_data	Bit based stack Compression - initially empty, finally compressed header and payload Decompression - initially compressed header, finally empty
unc_fields	Bit based stack Compression - initially empty, used to store fields to be sent uncompressed for transfer to compressed_data Decompression - initially data which has been sent uncompressed, finally only payload for transfer to uncompressed_data
received_data	Bit based stack Decompression - initially packet received, splits into compressed_data and unc_fields
control_data	Item based stack Compression and Decompression - initially and finally empty - storage for control fields if not used immediately after generation
u_flag	A flag which starts with value zero. It keeps track of which stack (compressed_data or unc_fields) has the compressed data added to/ removed from it.
compression_stack	Pointer to either compressed_data or unc_fields, whichever is the one that is currently having data put onto it according to the value of u_flag. Initially at both compression and decompression

compression_stack = compressed_data.

context	Storage of data as reference for compression and decompression - referenced by 'enc_index'
storage	Storage of data during (de)compression to be transferred to context if successful - referenced by 'enc_index'
enc_index	A counter to keep track of the field encodings and context data associated with them
method_chosen	List of methods used to compress a given header - lists the encoding method chosen for each 'enc_index'
compressor_state	The current state at the compressor (can be set to "IR", "IR-DYN" or "CO")
current_set	The set of compressed header formats currently in use
crc_static	The static part of the header
crc_dynamic	The dynamic part of the header
crc	The decompressed crc for checking against one calculated over full header (a width defined value)
MSN	The Master Sequence Number - a width defined value (its value increases by one for each packet received at the compressor or it is set to be a specific value using the SET-MSN method if the header contains a suitable field)
msn_bits	The number of bits chosen to encode the MSN
msn_lsbs	The LSBs of MSN used for padding (a width defined value)

A.1 Compressor

This section describes the EPIC-LITE header compressor.

A.1.1 Step 1: Packet classification

The input to the EPIC-LITE compressor is simply an uncompressed

packet. The compressor does not know whether the packet contains an RTP header, TCP header or other type of header, and hence the first step is to determine which (if any) ROHC context can be used to compress the packet.

With any profile generated by EPIC-LITE the packet classification is performed automatically, since the profile will reject any packet that it cannot successfully compress relative to the chosen context.

Note however that additional packet classification MAY be performed before the packet is passed to the EPIC-LITE compressor. For example the compressor MAY wish to check that the values that are initialised in the IR packet, or fixed for all states take the values specified in the prospective context before compression is attempted, as if they do not then compression will not succeed.

A.1.2 Step 2: Using the state machine

The job of the state machine is to choose whether to send IR, IR-DYN or compressed (CO) packets. Since EPIC-LITE currently operates in a unidirectional mode of compression only there is no need to synchronize the decision with the decompressor, and hence the choice can be left as an implementation decision.

A.1.3 Step 3: Compressing the header

The next step is to choose the compressed header format that will be used to transmit the header from the compressor to the decompressor. Given the selected profile the compressor has exactly max_sets + 2 possible sets of header formats available: a total of max_sets different sets of CO packets, as well as a set of IR-DYN packets and a set of IR packets. The choice of which header formats to use

depends on the current state of the state machine.

The compressor calls the function COMPRESS to compress the header. The function takes one parameter - the name of the method that is currently being used. The global variable enc_index is initially 1.

Note that is not necessary to provide EPIC-LITE with a description of where the fields occur in the uncompressed header, as this information is provided automatically as part of each method (written in the BNF input language). Each encoding method has a specific number of bits associated with it which is the number of bits that encoding method can compress - this splits the header into fields.

The function has a single output: a Boolean value indicating whether or not compression has successfully occurred. Additionally, if compression succeeds then the stack compressed_data will contain the

compressed value of every field in the uncompressed header.

The function also modifies the value of the global variable, method_chosen. If compression is successful then for each field in the profile, method_chosen will contain an indicator of which encoding method has been selected for the field. This is mapped onto a set of indicator flags in Section A.1.4.

The compression commences with a call to the COMPRESS function for the method specified in CO_packet (or IR_DYN_packet or IR_packet depending on the compressor state).

The function may call itself recursively with a different input.

For one particular aspect of profile complexity there are two distinct categories. For a profile in the first category, if a header is compressible, then compression is guaranteed to complete successfully in a single pass. The second category of profiles cannot make this guarantee. This is due either to a change of packet format (as indicated through the FORMAT encoding method) or to the failure of a non-library encoding method. A change of format requires IR-DYN packets to be sent indicating the change. Multiple formats may need to be tried in order to find one that can successfully compress the packet, and the compressor should implement such a mechanism. Where a non-library encoding method fails,

alternative encoding methods may be available. However, some encoding methods may have been applied, altering the contents of the stacks used to store state information for compression. The compressor should implement a mechanism, such as rolling back the compression state information, to allow alternative encoding methods to be tried. This ensures that all possible combinations of encoding methods can be tried to find one which will successfully compress the packet.

If the method is specified as part of the EPIC-LITE library, the pseudocode for the COMPRESS procedure is specified separately (in Appendix A.4.).

```
function COMPRESS (method_name)

    var enc, method
    var compress_function

    enc = first-field (method_name)

    while (enc <> NULL_ENCODING)
```



```
      method = first-method (enc)

      do
        # compress_function will be for the method if it is a
        # library method or a recursive call to this function for
        # a composite method...



        compress_function = lookup-compress-function
                                    (extract-name (method))

        can_compress = call compress_function
                                    (extract-name (method))

        if can_compress = true then

          # store the method selected from this field_encoding in
```

```
            # method_chosen
           store (method_chosen, method, enc)
          else
            # otherwise try another method in the field_encoding
            method = next-method (enc)
          endif
        loop until (can_compress = true) or (method = NULL_METHOD)
        if can_compress = true then
          enc = next-field (method_name)
        else
          return false
        endif
      end while
      return true
  end COMPRESS
```


N.B. This procedure shows the encoding methods for a field being checked in the order in which they appear in the profile. The order in which they are checked is actually implementation specific but is shown in the form above for simplicity of the pseudocode. But, for interoperability the "store" function must have a unique mapping dictated by the order of the encoding methods in the profile between the field and the encoding method.

Note that once the header has been compressed, the variable "storage" should contain the uncompressed value of each field which has context associated with it. This information is then transferred to the

context in the compressor (possibly overwriting one of the copies of information already stored).

A.1.4 Step 4: Determining the indicator flags

The next step is to determine the correct indicator flags for the chosen compressed header format.

Once the indicator flags have been added the header should be padded to be a multiple of bit_alignment bits and the uncompressed fields and payload added.

The compressor must run the following procedures:

```
function get-indicator-flags (method_chosen, flags_list_elt, Mvalue)
    var main_list_item, old_item, ml
    var last_N_before_item, w, length, length_item, next_length, num
    var last_N_of_smaller_length
    var found_length, n_diff
    var fl

    # How this mapping is done is implementation dependent at the
    # compressor

    get-format (main_list_item, method_chosen)

    old_item = next-item (main_list_item)
    last_N_before_item = tail-of (old_item.cum_N_list)

    w = head-of (main_list_item.cum_N_list)
    length = head-of (main_list_item.length_list)
    found_length = 0

    # If length_list only has one item then this loop will only
    # be done once (as in EPIC-LITE), otherwise loop until the
    # cumulative_N which covers Mvalue is found.  This gives the
```



```
    # length of the flags

    while (found_length = 0)
      n_diff = w - last_N_before_item
```

```
      if (n_diff < Mvalue) then
        w = next-item (w)
        length = next-item (length)
      else
        found_length = 1
      endif
    end loop

    # Find first flags of this length from the flag list

    found_length = 0
    fl = head-of (flag_list)

    while (found_length = 0)
      if length = fl.length then
        found_length = 1
      else
        fl = next-item (fl)
      endif
    end loop

    # Find the last cumulative N which has flags of smaller length
    # than fl

    found_length = 0
    ml = tail-of (main_list)

    while (found_length = 0)
      if (head-of (ml.length_list) < length) then
        ml = previous-item (ml)
      else
        found_length = 1
        ml = next-item (ml)
      endif
    end loop

    length_item = head-of (ml.length_list)
    num = head-of (cum_N_list)
    next_length = next-item (length_item)
    found_length = 0

    while (next_length <> NULL and found_length = 0)
      if next_length <> length then
        length_item = next-item (length_item)
```

```
          next_length = next-item (next_length)
          num = next-item (num)
        else
          found_length = 1
        endif
      end loop

      last_N_of_smaller_length = num

      # Work out what the indicator flags will be for this Mvalue

      flags = Mvalue + fl.flags + last_N_before_item -
                       last_N_of_smaller_length

      return length
  end get-indicator-flags


  procedure INDICATOR-FLAGS

      var n, bit, temp, length
      var extra_bits

      # Take account of the currently selected FORMAT set

      choose (main_list or formats and associated flag_list according
              to the value of current_set)

      length = get-indicator-flags (method_chosen, flags, 0)

      push (compressed_data, length, flags)

      # pad the compressed header to bit_alignment with extra bits of
      # MSN - use zeros if more space than bits of MSN
      bit = bit_alignment

      n = (bit - (stack-size (compressed_data) mod bit)) mod bit

      temp = value (MSN) - value (lsb (MSN, msn_bits))
      temp = temp / (2^msn_bits)
      extra_bits = lsb (temp, n)

      push (unc_fields, extra_bits)

      # add the uncompressed fields after the compressed header
      add (compressed_data, unc_fields)
```

```
        # add the payload after the uncompressed fields
```



```
        add (compressed_data, uncompressed_data)

    end INDICATOR-FLAGS
```

A.1.5 Step 5: Encapsulating in ROHC packet

The last step is to encapsulate the EPIC-LITE compressed packet within a ROHC packet.

Note that this includes adding the CID and any other ROHC framework headers (segmentation, padding etc.) as described in ROHC [1]. The ROHC packet is then ready to be transmitted.

A.2 Decompressor

This section describes the EPIC-LITE header decompressor.

A.2.1 Step 1: Decapsulating from ROHC packet

The input to the EPIC-LITE decompressor is a compressed ROHC packet. The first step is to read the CID of the packet and to extract the EPIC-LITE packet for parsing by the appropriate profile.

If the ROHC packet is identified as containing an EPIC-LITE compressed packet then the decompression process continues as indicated below.

A.2.2 Step 2: Running the state machine

The decompressor state machine determines whether the received packet should be decompressed or discarded (based on whether the decompressor believes its stored context to be valid or invalid). Since the compressor and decompressor state machines do not have to be synchronized, this is left as an implementation decision.

A.2.3 Step 3: Reading the indicator flags

The input to Step 3 is an EPIC-LITE compressed packet. Note that the

overall length of the packet is known from the link layer, but the length of the compressed header itself is NOT known.

The first step is to determine the compressed header format and split the packet into compressed header and uncompressed data. This is accomplished by reading the indicator flags as per the following procedure:


```
procedure decode-indicator-flags (received_data, method_chosen,
                                  flags, bit_chunks, Mvalue)

    var main_list_item, ml
    var last_N_before_item, length, length_item, next_length, num
    var last_N_of_smaller_length
    var found_length
    var fl

    found_length = 0
    fl = head-of (flag_list)

    while (found_length = 0 and fl <> NULL)
      if (top (received_data, fl.length*bit_chunks)
                                    <= fl.flags) then
        found_length = 1
      else
        fl = next-item (fl)
      endif
    end loop

    if fl <> NULL then
      fl = previous=item (fl)
    else
      fl = tail-of (flag_list)
    endif

    length = fl.length
    flags = top (received_data, length*bit_chunks)

    # Find the last cumulative N which has flags of smaller length
    # than fl
```

```
found_length = 0
ml = tail-of (main_list)

while (found_length = 0)
  if (head-of (ml.length_list) < length) then
    ml = previous-item (ml)
  else
    found_length = 1
    ml = next-item (ml)
  endif
end loop

length_item = head-of (ml.length_list)
num = head-of (ml.cum_N_list)
next_length = next-item (length_item)
```



```
found_length = 0

while (next_length <> NULL and found_length = 0)
  if next_length <> length then
    length_item = next-item (length_item)
    next_length = next-item (next_length)
    num = next-item (num)
  else
    found_length = 1
  endif
end loop

last_N_of_smaller_length = num

Mvalue = flags - fl.flags

num = tail-of (ml.cum_N_list)

while ((num - last_N_of_smaller_length) < Mvalue)
  ml = previous-item (ml)
  num = tail-of (ml.cum_N_list)
end loop

main_list_elt = ml
ml = next-item (ml)
last_N_before_item = tail-of (ml.cum_N_list)
```

```
        Mvalue = Mvalue + last_N_of_smaller_length - last_N_before_item

        # How this mapping is done is decompressor specific
        get-method-list (main_list_item, method_chosen)

    end decode-indicator-flags


    procedure READ-FLAGS
        var found, n, size, bit, k, len, Mvalue
        var temp, flags, flags_temp

        # Take account of the currently selected FORMAT set

        choose (main_list or formats and associated flag_list according
                to the value of current_set)

        decode-indicator-flags (received_data, method_chosen,
                                flags, 1, Mvalue)

        len = length (flags)
```



```
        pop (received_data, len, flags_temp)

        n = get-header-length (method_chosen)

        # put the compressed header on the compressed data stack
        pop (received_data, n, temp)
        push (compressed_data, temp)

        # store any extra lsbs of MSN sent
        bit = bit_alignment
        k = (bit - (n + len) mod bit) mod bit
        pop (received_data, k, msn_lsbs)

        # put the rest of the received packet on the unc_fields stack
        size = stack-size (received_data)
        pop (received_data, size, temp)
        push (unc_fields, temp)

    end READ-FLAGS
```

A.2.4 Step 4: Decompressing the fields

Now that the format of the compressed header has been determined, the next step is to decompress each field in turn.

The decompressor calls the procedure DECOMPRESS to calculate the uncompressed value of the fields. The only input to the procedure is the name of a method. Unlike the COMPRESS function there are no outputs since decompression always succeeds (although if the packet is corrupted, the correct answer may not be obtained).

Initially, the DECOMPRESS procedure is called for the method specified in CO_packet (or IR_DYN_packet or IR_packet depending on the ROHC packet type received). Note that as for COMPRESS the procedure may call itself recursively with different inputs. The global variable enc_index is initially 1.


```
procedure DECOMPRESS (method_name)

    var enc, method
    var decompress_function

    enc = last-field (method_name)

    while (enc <> NULL_ENCODING)

      method = get-method (method_chosen, enc)

      # decompress_function will be for the method if it is a
```

```
        # library method or a recursive call to this function for
        # composite method...

        decompress_function = lookup-decompress-function
                                        (extract-name (method))

        call decompress_function (extract-name (method))

        enc = prev-field (method_name)

      end while

  end DECOMPRESS
```

Observe that the DECOMPRESS procedure reads the input code in the opposite order to the COMPRESS procedure. This is because decompression is the exact mirror-image of compression: if fields are parsed in reverse order then it will never be the case that a field can only be decompressed relative to a field that has not yet been reached.

When the entire header has been decompressed it is on the stack uncompressed data and the payload is still on the stack unc_fields. These should be combined to make the original packet.

A.2.5 Step 5: Verifying correct decompression

By this stage the decompressor has calculated the value uncompressed_data that contains the entire uncompressed header as well as the payload.

The final step is to verify that successful decompression has occurred by applying the checksum to the uncompressed header. The CRC method makes available the variables checksum_value (containing the checksum from the compressed header) and crc_static + crc_dynamic

(containing all of the fields in the uncompressed header). The CRC should be evaluated over crc_static + crc_dynamic and compared with the CRC stored in checksum_value.

If the uncompressed header fails the checksum then it should be

discarded. If it passes then it can be forwarded by the decompressor.

Furthermore, if decompression is successful then the values contained within context can be replaced by the values contained in storage.

A.3 Offline processing

This section describes how the profile is converted into one or more sets of compressed header formats. Note that the following algorithms are run once offline and the results stored at the compressor and decompressor.

A.3.1 Step 1: Building the header formats

The first step is to build up a list of the max_formats different compressed header formats that occur with the highest probability (based on the probability values given in the input code).

To generate the max_sets + 2 different sets of compressed header formats, the BUILD procedure is called max_sets times with the global variable compressor_state set to "CO" and current_set taking values from 0 to max_sets - 1 inclusive. Additionally it is called once with compressor_state = "IR" and once with compressor_state = "IR-DYN". The output in each case is a list describing the top max_formats different compressed header formats. The list has the following attributes:

The output from the BUILD procedure is a list describing the top max_formats different compressed header formats in the following way.

Each list_item has several parts:

list_item.P	Overall percentage probability that the header format identified in this list item will be used
list_item.N	Total size of the header format identified in this list item in bits, excluding indicator flags
list_item.id	A list of integers uniquely identifying the header format associated with this list item (id is itself a list on which the list functions defined earlier can be performed)

Note that all percentages are stored to exactly 2 decimal places (or by scaling they can be stored as a 2-octet integer from 0 to 10000 inclusive). When two percentages are multiplied, the result MUST be calculated exactly (i.e. to 4 decimal places, or equivalently a 4-octet integer) and then truncated to 2 decimal places.

For example, 15.8% is represented as 1580, 89.2% is represented as 8920. The result of multiplying these two values together is:
1580 x 8920 = 14093600 (interim result, accurate to 4 DP)
14093600 / 10000 = 1409 (any remainder is discarded)
1409 = 14.09%

For interoperability the top max_formats entries MUST NOT be reordered when the discarding process is carried out. In the event of a tie, the list entries with the lowest indices are kept.

Note that the procedure may call itself recursively using a different input.

Some functions used in the BUILD procedure are defined first:

```
#
# compare two items by probability
#
function COMPARE (a, b)
    if a.P > b.P then
      return 1
    else if a.P < b.P then
      return -1
    else
      return 0
    endif
end COMPARE
```

```
#
# discard items from list such that:
#   list is sorted
#   if list contains less than max_formats entries then
#      the sorted list is returned


#   else if list contains more than max_formats then
#      keep the max_formats most probably entries
#      if any other entries have the same probability as
#      the least probable entry, keep those
#      discard the rest
#
procedure DISCARD (list, max_formats)

    var result_list
    var count
    var last_item

    empty (result_list)

    sort-natural (list, COMPARE)

    count = 0
    last_item = tail-of (list)

    foreach item in reverse list

      if ((count >= max_formats) and ((last_item.P <> item.P) or
                                      (item.P = 0)) then
        break
      endif

      prepend (result_list, item)

      last_item = item

      count = count + 1

    end loop

    copy (list, result_list)
```

```
   end DISCARD
```



```
   #
   # combine two lists by creating their
   # product. This sums the N values and
   # multiplies the P values
   #
   procedure COMBINE (final, temp)

       empty (new_list)

       foreach dst in temp

         foreach src in final

           item.P = dst.P * src.P
           item.P = dst.N + src.N
           copy (item.id, dst.id)

           concatenate (item.id, src.id)

           append (new_list, item)

         end loop

         DISCARD (new_list, max_formats)

       end loop

       copy (final, new_list)

   end COMBINE



   #
```

```
   # build the list for the method given
   #
   procedure BUILD (method_name, probability, build_list)

       var enc, method
       var item
       var final_list, build_output, temp_list
       var build_function

       enc = first-field (method_name)

       item.P = convert-percentage (100%)
       item.N = 0
```



```
       empty (item.id)

       empty (final_list)
       append (final_list, item)

       while (enc <> NULL_ENCODING)

         empty (temp_list)

         method = first-method (enc)

         do
         # build_function will be for the method if it is a library
         # method or a recursive call to this function for a composite
         # method...

           empty (build_output)

           build_function = lookup-build-function (method)

           call build_function (extract-name (method),
                                extract-probability (method),
                                build_output)

           foreach item in build_output

             append (item.id, method)
             append (temp_list, item)
```

```
            end loop
            DISCARD (temp_list)
            method = next-method (enc)
          loop until method = NULL_METHOD
          # combine lists - result in final_list
          COMBINE (final_list, temp_list)
          enc = next-field (method_name)
        end while
        foreach item in final_list
          item.P *= probability
```



```
        end loop
        copy (build_list, final_list)
    end BUILD


    procedure BUILD-TABLE (base_method, result)
        BUILD (base_method, convert-percentage (100%), result)
    end BUILD-TABLE
```

The final output of BUILD is a list describing max_formats different compressed header formats.

A.3.2 Step 2: Generating the indicator flags

The final step of generating a new set of compressed header formats is to convert the list of probabilities into a set of indicator flags. Each header format begins with a unique pattern of indicator flags that serve to distinguish it from all other header formats in the set.

EPIC-LITE generates the indicator flags using ordinary Huffman compression. For each of the cases in Section A.3.1 where the BUILD algorithm is run the following algorithm should be applied to the output of BUILD:

```
#
#   build the flags associated with the list given, reserving the
#   bit pattern '111' if do_reserve is set
#
procedure BUILD-FLAGS (main_list, do_reserve)

    var work_list, reserve
    var u, v, w, z
    var i, flag, value, prev_length, num_formats

    sort-natural (main_list, COMPARE)

    DISCARD (main_list, max_formats)

    empty (work_list)
```



```
    u = head-of (main_list)
    v = head-of (work_list) # which will be null

# the work_list starts empty, so v is 'null'. However, as soon as
# the first item is added to the work list, v references this item,
# which is why there is a condition & re-initialisation of v at the
# bottom of the loop

    item.P = u.P
    u.parent = item
    num_formats = size-of (main_list)

    for i = 0 to (num_formats - 2)
```

```
        if (i = (num_formats - 4) and do_reserve) then
          RESERVE (u, v, reserve, flag)
        endif

        u_next = next-item (u)
        v_next = next-item (v)

        if (at-end (v) or (
            not at-end (u_next) and (u_next.P <= v.P)) then

          item.P = u.P + u_next.P
          append (work_list, item)
          u.parent = item
          u_next.parent = item
          u = next-item (u_next)

        else if (at-end (u) or (
                 not at-end (v_next) and v_next.P <= u.P)) then

          item.P = v.P + v_next.P
          append (work_list, item)
          v.parent = item
          v_next.parent = item
          v = next-item (v_next)

        else

          item.P = u.P + v.P
          append (work_list, item)
          u.parent = item
          v.parent = item


          u = u_next
```



```
          v = next-item (v)

        endif

        if (i = 0) then

          v = head-of (work_list)
```

```
      endif

    end loop

    item.parent = NULL_ITEM
    cumulative_N = 0
    empty (flag_list)
    w = tail-of (main_list)

    for i = 0 to (num_formats - 1)

      z = w
      length = 0

      do
        if do_reserve then
          if (type = 0 and (z = reserve [0])) then
            length = length + 1

          else if (type = 1 and (z = reserve [0] or z = reserve [1])) then
            length = length + 1

          else if (type = 2 and z = reserve [3]) then
            length = length + 1

          else if (type = 2 and z = reserve [1]) then
            length = length - 1

          else if (type = 3 and z = reserve [2]) then
            length = length + 1
          endif

        endif

        length = length + 1
        z = z.parent

      loop until z.parent = NULL_ITEM

      cumulative_N = cumulative_N + 1
```

```
  append (w.length_list, length)
  append (w.cum_N_list, cumulative_N)

  # Add this length into the flag_list - either by adding
  # 1 to the element with correct length or by adding a
  # new element. Flag_list should be in ascending order of
  # lengths.

  found_same_length = 0
  fl_elt = head-of (flag_list)

  while (fl_elt <> NULL and found_same_length = 0)
    if fl_elt.length = length then
      found_same_length = 1
    endif
  end while

  if found_same_length = 0 then
    fl_elt.N = cumulative_N
    fl_elt.length = length
    append (flag_list, fl_elt)
  else
    fl_elt.N = fl_elt.N + 1
  endif

  w = previous-item (w)

end loop

# Generate the Huffman flags

flag_item.N = 0
flag_item.length = 0
flag_item.flags = 0
prepend(flag_list,flag_item)

fl_3 = head-of (flag_list)
fl_2 = next-item (fl_3)

fl_2.flags = str(fl_2.length,0)
fl_1 = next-item(fl_2)

while (fl_1 <> NULL)

   fl_1.flags = str(fl_1.length,(fl_2.flags + (fl_2.N - fl_3.N))
                    * 2 ^ (fl_1.length - fl_2.length))

   fl_1 = next-item (fl_1)
```

```
          fl_2 = next-item (fl_2)
          fl_3 = next-item (fl_3)

       end while

   end BUILD_FLAGS




   #
   #   procedure to work out whether to use one or two list
   #   items to reserve the bits '111'
   #
   procedure RESERVE (u, v, reserve, flag)

       var temp_u, temp_v
       var t
       var prob

       temp_u = u
       temp_v = v

       flag = 0


       # This section works out which order the 4 remaining nodes
       # will be formed into tree
       for t = 0 to 3

         if (at-end (temp_v) or (
             not at-end (temp_u) and temp_u.P <= temp_v.P) then

           reserve [t] = temp_u
           prob [t] = temp_u.P
           temp_u = next-item (temp_u)

         else

           reserve [t] = temp_v
           prob [t] = temp_v.P
           temp_v = next-item (temp_v)
```

```
        endif

      end loop

      # prob[0] = A ... prob [3] = D
```



```
      # This function is pretending to create a node which will
      # have the same probability as B and will take up the bit
      # pattern '111'.  If the new node were to be created then
      # check whether a + 2*B >= D to decide whether or not the
      # node comes before or after B.  This part looks at the
      # probabilities to work out which nodes would have their
      # lengths changed by inserting this node so they can be
      # changed accordingly in the length counting (rather than
      # actually trying to add the new node)

      if ((prob [0] + prob [1]) >= prob [3]) then
        flag = 0
      else
        if ((prob [0] + prob [1]) < prob [2]) then
          if ((prob [0] + (2 * prob [1])) < prob [3]) then
            flag = 1
          else
            flag = 2
          endif
        else
          if ((prob [1] + prob[2] < prob [3])) then
            flag = 3
          else
            flag = 2
          endif
        endif
      endif

  end RESERVE
```

The procedure RESERVE is called at most once by BUILD-FLAGS to reserve the bit pattern "111" in the first octet of each compressed header (for compatibility with the ROHC framework).

The output of BUILD-FLAGS is the main_list each item of which has

three key parts:

list_item.length_list	List of lengths of numbers of flags (for EPIC-LITE this will only have one element). The lengths should (automatically) be in ascending order.
list_item.cum_N_list	A list of numbers - corresponding to the number of headers with each flag length in the length_list (again for EPIC-LITE there will only be one element). The numbers should (automatically) be in ascending order.
list_item.identifier	Some unique identifier which can be used to perform mapping between these flags and the header format used to generate a compressed header requiring these flags.

and a flag_list each item of which has the following elements:

flag_list_item.length	The length of the Huffman flags.
flag_list_item.N	The cumulative number of headers with Huffman flags up to this length.
flag_list_item.flags	The flags for the first header with flags of this length (the rest can be worked out from this information).

Note that the compressor assigns bit patterns to the indicator flags

using the following rules:

1. The most probable headers have all "0" indicator flags

2. The indicator flags for the next header format are calculated by adding 1 to the previous flags (treated as an integer) and padding with enough 0s to reach the correct length

As an example, the indicator flags for a set of compressed header formats are given below:

main_list

Compressed header format	No. of flags	Cumulative_N
1	2	1
2	2	2
3	3	3
4	4	4
5	4	5
6	4	6
7	5	7
8	6	8
9	6	9

flag_list

No. of flags	N	Bit pattern of flags
2	1	00
3	3	100
4	4	1010
5	7	11010
6	8	110110

Note that the most probable compressed header format will have all

"0" indicator flags, whilst the least probable header format will have all "1" indicator flags (except for the bit pattern "111" if this is reserved for the ROHC [1] framework).

A.4 Library of methods

This section gives pseudocode for each of the methods in the library. Note that for each method three pieces of pseudocode are given: corresponding to the function COMPRESS, and procedures DECOMPRESS and BUILD described previously.

Note that all of the global variables required for these procedures are defined at the beginning of Appendix A.

It is assumed that as soon as the 'return' command is encountered, the procedure stops.

For COMPRESS functions which check through the r values stored in the context, the value of r is implementation-specific. Note that if any of the r context values is empty, any method attempting to compress relative to the context will automatically fail.

A.4.1 STATIC

STATIC (P%)


```
COMPRESS (STATIC)
    var context_val, item
    var n, i

    context (enc_index, 1, context_val)
    n = length (context_val)

     # check that the value to be compressed matches each of the r
     # values stored in context for this encoding - if not then
     # STATIC can't be used to compress this encoding

    for i = 1 to r
      context (enc_index, i, context_val)
      if (context_val <> top (uncompressed_data, n)) then
        return false
```

```
      endif
    end loop

    pop (uncompressed_data, n, item)
    save (enc_index, item, storage)
    enc_index = enc_index + 1
    return true
end COMPRESS


DECOMPRESS (STATIC)
    var context_val

    context (enc_index, 1, context_val)
    push (uncompressed_data, context_val)
    save (enc_index, context_val, storage)
    enc_index = enc_index + 1
end DECOMPRESS


BUILD (STATIC, P, format)
    var item

    item.P = P
    item.N = 0
    empty (item.id)
    append (format, item)
end BUILD
```

A.4.2 IRREGULAR


```
IRREGULAR(n,P%)

COMPRESS (IRREGULAR)
    var item

    pop (uncompressed_data, n, item)
    push (compression_stack, item)
    save (enc_index, item, storage)
```

```
        enc_index = enc_index + 1
        return true
    end COMPRESS


    DECOMPRESS (IRREGULAR)
        var item

        pop (compression_stack, n, item)
        push (uncompressed_data, item)
        save (enc_index, item, storage)
        enc_index = enc_index + 1
    end DECOMPRESS


    BUILD (IRREGULAR, P, format)
        var item

        item.P = P
        item.N = n
        empty (item.id)
        append (format, item)
    end BUILD
```

A.4.2.1 IRREGULAR-PADDED

```
    IRREGULAR-PADDED(n,k,P%)

    COMPRESS (IRREGULAR-PADDED)
        var item, temp

        if (value (top (uncompressed_data, n - k)) <> 0) then
          return false
        else
          pop (uncompressed_data, n, item)
          temp = lsb (item, k)
          push (compression_stack, temp)
```



```
          save (enc_index, item, storage)
```

```
        enc_index = enc_index + 1
        return true
      endif
  end COMPRESS


  DECOMPRESS (IRREGULAR-PADDED)
      var item, temp
      var full

      full = 0
      pop (compression_stack, k, temp)
      full = full + value (temp)
      item = str (n, full)
      push (uncompressed_data, item)
      save (enc_index, item, storage)
      enc_index = enc_index + 1
  end DECOMPRESS


  BUILD (IRREGULAR-PADDED, P, format)
      var item

      item.P = P
      item.N = k
      empty (item.id)
      append (format, item)
  end BUILD
```

A.4.3 VALUE

```
VALUE(n,v,P%)

COMPRESS (VALUE)
    var value

    if (top (uncompressed_data, n) <> v) then
      return false
    else
      pop (uncompressed_data, n, value)
      save (enc_index, value, storage)
      enc_index = enc_index + 1
      return true
    endif
end COMPRESS


DECOMPRESS (VALUE)
    var item

    item = str (n, v)
    push (uncompressed_data, n, v)
    save (enc_index, item, storage)
    enc_index = enc_index + 1
end DECOMPRESS


BUILD (VALUE, P, format)
    var item

    item.P = P
    item.N = 0
    empty (item.id)
    append (format, item)
end BUILD
```

A.4.4 LSB

```
LSB(k,p,P%)

COMPRESS (LSB)
    var context_val, item, temp, new_item, lsb_val, p_item
    var n, i
```

```
        context (enc_index, 1, context_val)
        n = length (context_val)
```



```
        p_item = str (n, p)
        new_item = top (uncompressed_data, n)
        for i = 1 to r
          context (enc_index, i, context_val)

          # check new_item in interval
          # [value (context_val)-p, value (context_val)-p+2^k]

          temp = (new_item - context_val + p_item)
          if (value (temp) < 0) or (value (temp) >= 2^k) then
            return false
          endif
        end loop

        pop (uncompressed_data, n, item)
        lsb_val = lsb (item, k)
        push (compression_stack, lsb_val)
        save (enc_index, item, storage)
        enc_index = enc_index + 1
        return true
    end COMPRESS


    DECOMPRESS (LSB)
        var context_val, interval_start, interval_end, recd, p_item
        var new_item, twok_item
        var n, new, start, end

        context (enc_index, 1, context_val)
        n = length (context_val)
        p_item = str (n, p)
        twok_item = str (n, 2^k)

        pop (compression_stack, k, recd)
        interval_start = context_val - p
        interval_end = interval_start + twok_item
        new_item = concat (msb (interval_start, (n-k)), recd)

        # check whether value (new_item) is in interval
```

```
        # [value (interval_start), value (interval_end)]
        # allowing for the interval to wrap over zero. If not then
        # recalculate new_item

        start = value (interval_start)
        end = value (interval_end)
        new = value (new_item)

        if (((start < end) and ((new < start) or (new > end))) or
```



```
            ((start > end) and ((new > start) or (new < end)))) then

          new_item = concat (msb (interval_end, (n-k)), recd)

        endif

        push (uncompressed_data, new_item)
        save (enc_index, new_item, storage)
        enc_index = enc_index + 1
    end DECOMPRESS


    BUILD (LSB, P, format)
        var item

        item.P = P
        item.N = k
        empty (item.id)
        append (format, item)
    end BUILD
```

A.4.5 UNCOMPRESSED

```
    UNCOMPRESSED(n,d,m,p)

    COMPRESS (UNCOMPRESSED)
        var scale_len
        var unc, len_item
```

```
        scale_len = floor( value((top(control_data)) / d) * m + p)
        if (stack-size (uncompressed_data) < scale_len) then
          return false
        else
          pop (uncompressed_data, scale_len, unc)
          push (unc_fields, unc)
          pop (control_data, len_item)
          if (length (len_item) <> n) then
            return false
          endif
          push (uncompressed_data, len_item)
          return true
        endif
    end COMPRESS
```



```
    DECOMPRESS (UNCOMPRESSED)
        var scale_len
        var unc, len_item

        pop (uncompressed_data, n, len_item)
        scale_len = floor( (value (len_item) / d) * m + p)
        push (control_data, len_item)
        pop (unc_fields, scale_len, unc)
        push (uncompressed_data, unc)
    end DECOMPRESS


    BUILD
    end BUILD
```

A.4.6 STACK encoding methods

A.4.6.1 STACK-TO-CONTROL

```
    STACK-TO-CONTROL(n)

    COMPRESS (STACK-TO-CONTROL)
```

```
    var item

    pop (uncompressed_data, n, item)
    push (control_data, item)
    return true
end COMPRESS


DECOMPRESS (STACK-TO-CONTROL)
    var item

    pop (control_data, item)
    push (uncompressed_data, item)
end DECOMPRESS


BUILD (STACK-TO-CONTROL, 100%, format)
end BUILD
```


A.4.6.2 STACK-FROM-CONTROL

```
STACK-FROM-CONTROL(n)

COMPRESS (STACK-FROM-CONTROL)
    var item

    pop (control_data, item)
    if (length (item) <> n) then
      return false
    endif
    push (uncompressed_data, item)
    return true
end COMPRESS


DECOMPRESS (STACK-TO-CONTROL)
```

```
        var item

        pop (uncompressed_data, n, item)
        push (control_data, item)
    end DECOMPRESS


    BUILD (STACK-TO-CONTROL, 100%, format)
    end BUILD
```


A.4.6.3 STACK-PUSH-MSN

```
    STACK-PUSH-MSN(n)

    COMPRESS (STACK-PUSH-MSN)
        var temp

        temp = str (n, value (MSN) mod 2^n)
        push (control_data, temp)
        return true
```

```
end COMPRESS


DECOMPRESS (STACK-PUSH-MSN)
    var item

    pop (control_data, item)
end DECOMPRESS


BUILD (STACK-PUSH-MSN, 100%, format)
end BUILD
```


A.4.6.4 STACK-POP-MSN

```
STACK-POP-MSN(n)
```

```
COMPRESS (STACK-POP-MSN)
    var item, temp

    pop (uncompressed_data, n, item)
    temp = str (n, value (MSN) mod 2^n)

    if (item <> temp)
      #value on stack wasn't MSN
      return false
    endif
    return true
end COMPRESS


DECOMPRESS (STACK-POP-MSN)
    var temp

    temp = str (n, value (MSN) mod 2^n)
    push (uncompressed_data, temp)
end DECOMPRESS


BUILD (STACK-POP-MSN, 100%, format)
end BUILD
```

A.4.6.5 STACK-ROTATE

```
STACK-ROTATE(n,m)

COMPRESS (STACK-ROTATE)
    rotate (control_data, n, m)
    return true
end COMPRESS


DECOMPRESS (STACK-ROTATE)
    rotate (control_data, n, (n - m))
end DECOMPRESS


BUILD (STACK-ROTATE, 100%, format)
end BUILD
```

A.4.7 INFERRED encoding methods

A.4.7.1 INFERRED-TRANSLATE

```
INFERRED-TRANSLATE(n,m,a(0),b(0),a(1),b(1)...,a(k-1),b(k-1))

COMPRESS (INFERRED-TRANSLATE)
    var item, trans_item
    var found, i

    found = 0
    i = 0

    while ((i < k) and (found = 0))
      if (value (top (uncompressed_data, m)) = b(i))
        pop (uncompressed_data, m, item)
        trans_item = str (n, a(i))
        push (control_data, trans_item)
        found = 1
      else
        i = i + 1
      endif
    end while

    return found
```

```
   end COMPRESS
```



```
   DECOMPRESS (INFERRED-TRANSLATE)
       var trans_item
       var i, found

       found = 0
       i = 0

       pop (control_data, trans_item)
       while ((i < k) and (found = 0))
         if (value (trans_item) = a(i))
           push (uncompressed_data, m, b(i))
           found = 1
         else
           i = i + 1
         endif
       end while
   end DECOMPRESS


   BUILD (INFERRED-TRANSLATE, 100%, format)
   end BUILD
```

A.4.7.2 INFERRED-SIZE

```
INFERRED-SIZE(n,p)

COMPRESS (INFERRED-SIZE)
    var item
    var bits_in_byte # usually 8!

    if ((bits_in_byte * value (top (uncompressed_data, n))) + p) <>
      stack-size (uncompressed_data) then
      return false


    else
      pop (uncompressed_data, n, item)
      return true
    endif
end COMPRESS


DECOMPRESS (INFERRED-SIZE)
    var size
    var bits_in_byte # usually 8!

    size = (stack-size (uncompressed_data) + n - p) / bits_in_byte)
    push (uncompressed_data, n, size)
end DECOMPRESS


BUILD (INFERRED-SIZE, 100%, format)
end BUILD
```

A.4.7.3 INFERRED-OFFSET


```
   INFERRED-OFFSET(n)

   COMPRESS (INFERRED-OFFSET)
       var item, base, offset

       pop (uncompressed_data, n, item)
       pop (control_data, base)
       if (length (base) <> n) then
           return false
       endif

       offset = item - base
       push (uncompressed_data, offset)
       push (uncompressed_data, base)
       return true
   end COMPRESS


   DECOMPRESS (INFERRED-OFFSET)
       var item, base, offset

       pop (uncompressed_data, n, base)
       push (control_data, base)
       pop (uncompressed_data, n, offset)
       item = offset + base
       push (uncompressed_data, item)
```

```
   end DECOMPRESS


   BUILD (INFERRED-OFFSET, 100%, format)
   end BUILD
```

A.4.7.4 INFERRED-IP-CHECKSUM

```
   INFERRED-IP-CHECKSUM(new_method)

   COMPRESS (INFERRED-IP-CHECKSUM)
       var compress_function
       var can_compress
       var item, ip_sum

       # zero the ip checksum bits in the header
       pop (uncompressed_data, 80, item)
       pop (uncompressed_data, 16, ip_sum)
       push (uncompressed_data, 16, 0)
```



```
       push (uncompressed_data, item)

       compress_function = lookup-compress-function
                           (extract-name (new_method))

       can_compress = call compress_function
                           (extract-name (new_method))

       return can_compress
   end COMPRESS


   DECOMPRESS (INFERRED-IP-CHECKSUM)
       var decompress_function
       var item, temp
       var init_len, final_len, decomp_len, ip_sum

       init_len = stack-size (uncompressed_data)
       decompress_function = lookup-decompress-function
```

```
                                (extract-name (new_method))

        call decompress_function (extract-name (new_method))
        final_len = stack-size (uncompressed_data)

        decomp_len = final_len - init_len
        ip_sum = compute-16-checksum (uncompressed_data, decomp_len)

        pop (uncompressed_data, 80, item)
        pop (uncompressed_data, 16, temp)
        push (uncompressed_data, 16, ip_sum)
        push (uncompressed_data, item)
    end DECOMPRESS


    BUILD (INFERRED-IP-CHECKSUM, 100%, format)
        var build_function

        build_function = lookup-build-function
                                (extract-name (new_method))

        call build_function (extract-name (new_method), 100%, format)
    end BUILD
```

A.4.8 NBO


```
    NBO(n)

    COMPRESS (NBO)
        var original, swapped
        var nbo_flag

        pop (uncompressed_data, n, original)
        choose (nbo_flag of zero or one)
        # according to whether original is in network byte order or not
        # (1 implies it is, 0 implies it isn't)
        # this decision can be made based on the context history
```

```
        push (uncompressed_data, 1, nbo_flag)
        if (nbo_flag = 0) then
          swapped = byte-swap (original)
          push (uncompressed_data, swapped)
        else
          push (uncompressed_data, original)
        endif

        # store the original value or other implementation specific
        # information for making decision about flag value
        save (enc_index, original, storage)
        enc_index = enc_index + 1
        return true
    end COMPRESS


    DECOMPRESS (NBO)
        var original, decomped, nbo_flag

        pop (uncompressed_data, n, decomped)
        pop (uncompressed_data, 1, nbo_flag)

        if (value (nbo_flag) = 1 then
          original = decomped
        else
          original = byte-swap (decomped)
        endif

        push (uncompressed_data, original)
        enc_index = enc_index + 1
    end DECOMPRESS


    BUILD (NBO, 100%, format)
    end BUILD
```


A.4.9 SCALE

```
    SCALE(n)
```

```
COMPRESS (SCALE)
    var original
    var scale_f, scaled_val, remainder

    pop (uncompressed_data, n, original)
    choose (scale_f) # such that original is changing by a multile
                     # of scale_f each header.  This decision can
                     # be based on context history

    scaled_val = value (original) / scale_f
    remainder = value (original) mod scale_f

    push (uncompressed_data, n, scaled_val)
    push (uncompressed_data, n, remainder)
    push (uncompressed_data, n, scale_f)

    # store the original value or other implementation specific
    # information for making decision about scale_f
    save (enc_index, original, storage)
    enc_index = enc_index + 1
    return true
end COMPRESS


DECOMPRESS (SCALE)
    var scaled_val, scale_f, remainder
    var original

    pop (uncompressed_data, n, scale_f)
    pop (uncompressed_data, n, remainder)
    pop (uncompressed_data, n, scaled_val)

    original = ((scaled_val * scale_f) + remainder) mod 2^n
    push (uncompressed_data, n, original)
    enc_index = enc_index + 1
end DECOMPRESS


BUILD (SCALE, 100%, format)
end BUILD
```

A.4.10 OPTIONAL

```
OPTIONAL(new_method)

COMPRESS (OPTIONAL)
    var flag
    var compress_function
    var can_compress, n
    var enc

    pop (control_data, flag)
    if (value (flag) = 1) then
      compress_function = lookup-compress-function
                          (extract-name (new_method))

      can_compress = call compress_function
                          (extract-name (new_method))
    else
      choose (format to encode new_method)
      # compressor choice of format for new_method, where enc is the
      # format chosen

      if u_flag = 0 then
        # not under U method then pad with bits of zero
        n = count-bits (new_method, enc)
        push (compression_stack, n, 0)
      else
        # don't need to pad as under U method
      endif

      store (method_chosen, OPTIONAL, enc)
      can_compress = 1
    endif

    push (control_data, flag)
    return can_compress
end COMPRESS


DECOMPRESS (OPTIONAL)
    var flag, item
    var decompress_function
    var n
    var enc

    pop (control_data, flag)
    if (value (flag = 1)) then
```

```
        decompress_function = lookup-decompress-function
                                (extract-name (new_method))

        call decompress_function (extract-name (new_method))
      else
        # enc is the format sent for new_method (obtained from the
        # indicator flags)

        if u_flag = 0 then
          # not under U method so need to remove bits of padding
          n = count-bits (new_method, enc)
          pop (compression_stack, n, item)
        else
          # don't need to remove padding as under U method
        endif
      endif

      push (control_data, flag)
  end DECOMPRESS


  BUILD (OPTIONAL, 100%, format)
      var build_function

      build_function = lookup-build-function
                                (extract-name (new_method))

      call build_function (extract-name (new_method), P, format)

  end BUILD
```

A.4.11 MANDATORY

```
  MANDATORY(new_method)

  COMPRESS (MANDATORY)
      var flag
      var compress_function
      var can_compress
```

```
        pop (control_data, flag)
        if (value (flag) <> 1) then
          return false
        else
          compress_function = lookup-compress-function
```



```
                                (extract-name (new_method))

          can_compress = call compress_function
                                (extract-name (new_method))
        endif

        push (control_data, flag)
        return can_compress
    end COMPRESS


    DECOMPRESS (MANDATORY)
        var decompress_function
        var flag

        pop (control_data, flag)
        decompress_function = lookup-decompress-function
                                (extract-name (new_method))

        call decompress_function (extract-name (new_method))
        push (control_data, flag)
    end DECOMPRESS


    BUILD (MANDATORY, 100%, format)
        var build_function

        build_function = lookup-build-function
                                (extract-name (new_method))

        call build_function (extract-name (new_method), 100%, format)
    end BUILD
```

A.4.12 CONTEXT

```
   CONTEXT(new_method,k)

   COMPRESS (CONTEXT)
       var n, j, m
       var compress_function
       var can_compress

       n = ceiling (log2(k-1))
       choose (j < k) # compressor choice
       m = context-size (new_method)
```



```
       enc_index = enc_index + j * m
       compress_function = lookup-compress-function
                                (extract-name (new_method))

       can_compress = call compress_function
                                (extract-name (new_method))
       push (uncompressed_data, n, j)
       enc_index = enc_index + (k - j - 1) * m
       return can_compress
   end COMPRESS


   DECOMPRESS (CONTEXT)
       var n, j, m
       var decompress_function
       var index

       n = ceiling (log2(k-1))
       pop (uncompressed_data, n, index)
       j = value (index)

       m = context-size (new_method)
       enc_index = enc_index + j * m
       decompress_function = lookup-decompress-function
                                (extract-name (new_method))

       call decompress_function (extract-name (new_method))
       enc_index = enc_index + (k - j - 1) * m
```

```
end DECOMPRESS


BUILD (CONTEXT, 100%, format)
    var build_function

    build_function = lookup-build-function (new_method)

    call build_function (extract-name (new_method), 100%, format)
end BUILD
```

A.4.13 LIST

```
LIST(n,d,m,p,z,new_method(0),new_method(1),...,
            new_method(k-1),v(0),v(1),...,v(j))

COMPRESS (LIST)
```



```
    var scale_len, i, can_compress, stack_len, bits
    var comp_len, order, presence
    var present [0..(k-1)], unc_length
    var compress_function

    top(control_data, unc_length)
    scale_len = floor( value(unc_length) / d) * m + p)

    for i = 0 to (k - 1)
      present [i] = str (1,0)
    end loop

    order = 0
    bits = ceiling (log2(k-1))
    i = 0
    list_start_enc_index = enc_index

    while (scale_len > 0)


      # basically loop through the methods until all scale_len bits
```

```
        # are compressed and any of the methods which aren't used are
        # marked as not used. The order in which methods are checked
        # is implementation specific but some ways will require
        # changing the order more frequently (reducing efficiency) than
        # others.

        stack_len = stack-size (uncompressed_data)
        can_compress = false
        i = 0

        while (i < k) and (can_compress = false)
          if (value (present [i]) = 0) and
             ((i >= j) or ((i < j) and
              (value (top (uncompressed_data, z)) = v(i)))) then
            present [i] = str (1,1)
            order = order * 2^bits + i
            push (control_data, present [i])
            for x = 0 to i
              enc_index = enc_index + context-size (new_methods(x))
            compress_function = lookup-compress-function
                                (extract-name (new_method(i)))

            can_compress = call compress_function
                                (extract-name (new_method(i)))
            if can_compress = false then
              return false
            endif
```



```
            comp_len = stack_len - stack-size (uncompressed_data)
            scale_len = scale_len - comp_len
            pop (control_data, present [i])
            enc_index = list_start_enc_index
            if (length (present[i]) <> 1) then
              return false
            endif

          else
            i = i + 1
          endif
        end while
      end while
```

```
      # process remaining LIST entries that have not been used
      # (i.e. are not present in the header)

      for i = 0 to (k - 1)
        if (value (present [i]) = 0) then
          push (control_data, present [i])
          for x = 0 to i
            enc_index = enc_index + context-size (new_methods(x))
          order = order * 2^bits + i
          compress_function = lookup-compress-function
                              (extract-name (new_method(i)))

          can_compress = call compress_function
                              (extract-name (new_method(i)))
          if can_compress = false then
            return false
          endif
          pop (control_data, present [i])
          enc_index = list_start_enc_index

          if (length (present[i]) <> 1) then
            return false
          endif

        endif
      end loop

      presence = 0
      for i = 0 to (k - 1)
        presence = presence * 2 + value (present[i])
      end loop

      push (uncompressed_data, k, presence)
      push (uncompressed_data, bits*k, order)
```



```
      return true
   end COMPRESS


   DECOMPRESS (LIST)
      var decompress_function
      var presence, order, i, j, bits, uncomp_len, old_len,
```

```
      var original_len
      var present [0..(k - 1)]
      var presence_item, order_item

      bits = ceiling (log2(k-1))
      list_start_enc_index = enc_index

      pop (uncompressed_data, bits*k, order_item)
      pop (uncompressed_data, k, presence_item)
      presence = value (presence_item)
      order = value (order_item)

      for i = 0 to (k - 1)
        present [(k - 1) - i] = lsb (presence, 1)
        presence = (presence - value (present [(k - 1) - i])) / 2
      end loop

      for j = 0 to (k - 1)
        i = value (lsb (order, bits))
        order = (order û i )/ 2^bits

        push (control_data, present [i])
        old_len = stack-size (uncompressed_data)
        for x = 0 to i
          enc_index = enc_index + context-size (new_methods(x))
        decompress_function = lookup-decompress-function
                                (extract-name (new_method(i)))

        call decompress_function (extract-name (new_method(i)))

        enc_index = list_start_enc_index
        uncomp_len = uncomp_len +
                           (stack-size (uncompressed_data) - old_len)
        pop (control_data, present[i])

        original_len = ((uncomp_len - p) * d) / m
        push (control_data, n, original_len)
      end loop
   end DECOMPRESS
```

```
BUILD (LIST, 100%, format)
    var i
    var build_function
    var temp_format

    item.P = convert-percentage (100%)
    item.N = 0
    empty (item.id)
    append (format, item)

    for i = 0 to (k - 1)
      empty (temp_format)
      build_function = lookup-build-function (
                              extract-name (new_method(i)))
      call build_function (extract-name (new_method(i),
                              100%, temp_format))

      DISCARD (temp_format)
      COMBINE (format, temp_format)
    end loop
end BUILD
```

A.4.13.1 LIST-NEXT

```
LIST-NEXT (n,new_method(0),new_method(1),...,
            new_method(k-1),v(0),v(1),...,v(j))

# This pseudocode is very similar to LIST, the differences being
# from where the information about which method to use next comes
# from and how to tell when there is no more data to compress using
# this method

COMPRESS (LIST)
    var i, can_compress, bits, order, presence, p
    var present [0..(k-1)], v, null
    var compress_function

    for i = 0 to (k - 1)
      present [i] = str (1,0)
    end loop

    order = 0
    bits = ceiling (log2(k-1))
    i = 0
```

```
pop (control_data, v)
if (length (v) <> n) then
  return false
endif

null = str (0, 0)

while (v <> null)
  # basically loop through the methods until no more information
  # has been put on the control_data stack by the previous method
  # (i.e. the end of the list has been reached). The order in
  # which methods are checked is implementation specific but some
  # ways will require changing the order more frequently
  # (reducing efficiency) than others.

  can_compress = false
  i = 0

  while (i < k) and (can_compress = false)
    if (value (present [i]) = 0) and
       ((i >= j) or ((i < j) and (v = v(i)))) then
      present [i] = str (1,1)
      order = order * 2^bits + i
      push (control_data, present [i])
      p = stack-pointer (control_data)
      compress_function = lookup-compress-function
                          (extract-name (new_method(i)))

      can_compress = call compress_function
                          (extract-name (new_method(i)))
      if can_compress = false then
        return false
      endif

      # find out whether there is more to compress by checking
      # the position of the stack pointer

      if (p <> stack-pointer (control_data)) then
        pop (control_data, v)
        if (length (v) <> n) then
          return false
        endif
        pop (control_data, present [i])
```

```
            else
              pop (control_data, present [i])
              v = null
            endif
```



```
            if (length (present[i]) <> 1) then
              return false
            endif

          else
            i = i + 1
          endif
        end while
      end while

      # process remaining LIST-NEXT entries that have not been used
      # (i.e. are not present in the header)

      for i = 0 to (k - 1)
        if (value (present [i]) = 0) then
          push (control_data, present [i])
          order = order * 2^bits + i
          compress_function = lookup-compress-function
                              (extract-name (new_method(i)))

          can_compress = call compress_function
                              (extract-name (new_method(i)))
          if can_compress = false then


            return false
          endif
          pop (control_data, present [i])
          if (length (present[i]) <> 1) then
            return false
          endif

        endif
      end loop

      presence = 0
```

```
        for i = 0 to (k - 1)
          presence = presence * 2 + value (present[i])
        end loop

        push (uncompressed_data, k, presence)
        push (uncompressed_data, bits*k, order)
        return true
    end COMPRESS
```

DECOMPRESS is almost the same as for LIST - the only difference being that there is no need to keep track of length and push it onto the

control stack at the end as this is implicit in the choice of methods. BUILD is the same as for LIST

A.4.14 FLAG encoding methods

A.4.14.1 N

```
N(new_method)

COMPRESS (N)
    var compress_function
    var can_compress, temp

    temp = enc_index
    compress_function = lookup-compress-function
                            (extract-name (new_method))

    can_compress = call compress_function
                            (extract-name (new_method))


    if (enc_index <> temp) then
      clear (temp, storage)
    endif

    return can_compress
end COMPRESS
```

```
   DECOMPRESS (N)
       var decompress_function
       var temp

       temp = enc_index
       decompress_function = lookup-decompress-function
                                (extract-name (new_method))

       call decompress_function (extract-name (new_method))

       if (enc_index <> temp) then
         clear (temp, storage)
       endif
   end DECOMPRESS


   BUILD (N, (P = extract-probability (new_method)), format)
       var build_function
```



```
       build_function = lookup-build-function (method)

       call build_function (extract-name (new_method), P, format)
   end BUILD
```

A.4.14.2 U

```
   U(new_method)

   COMPRESS (U)
       var compress_function
       var can_compress

       if u_flag = 0 then
         compression_stack = unc_fields
       endif

       u_flag = u_flag + 1
       compress_function = lookup-compress-function
```

```
                              (extract-name (new_method))

      can_compress = call compress_function
                              (extract-name (new_method))

      u_flag = u_flag - 1
      if u_flag = 0 then
        compression_stack = compressed_data
      endif
      return can_compress
   end COMPRESS


   DECOMPRESS (U)
      var decompress_function

      if u_flag = 0 then
        compression_stack = unc_fields
      endif

      u_flag = u_flag + 1
      decompress_function = lookup-decompress-function
                              (extract-name (new_method))

      call decompress_function (extract-name (new_method))
      u_flag = u_flag - 1
```



```
      if u_flag = 0 then
        compression_stack = compressed_data
      endif
   end DECOMPRESS


   BUILD (U, (P = extract-probability (new_method)), format)
      var build_function

      build_function = lookup-build-function (method)

      call build_function (extract-name (new_method), P, format)

      foreach el in format
        el.N = 0
```

```
        end loop
    end BUILD
```

A.4.15 FORMAT

```
    FORMAT(new_method(0), ..., new_method(k - 1))

    COMPRESS (FORMAT)
        var n, can_compress, index_val
        var compress_function

        n = ceiling(log2(k-1))

        choose (index_val < k)
        # The compressor needs to make a choice on which FORMAT
        # to use.  The algorithm for making this choice is entirely
        # up to the compressor.  It may wish to take into account
        # compression performance (of compressed packets) and
        # flexibility of compression.  The selection processing
        # should allow a successful match to be made (if possible),
        # but must also terminate in the case where the packet is
        # incompressible.  It may be that a choice is made based on
        # "CO" state processing, which should be carried forward
        # into IR(-DYN) state packet generation.

        compress_function = lookup-compress-function (
                                extract-name(new_method(index_val)))

        can_compress = call compress_function (
                                extract-name (new_method(index_val)))
```



```
        if (can_compress <> false) then
          current_set = current_set * k + index_val
        endif

        if can_compress = false then
          return false
        else
          push (uncompressed_data, n, index_val)
```

```
        endif
        return true

    end COMPRESS


    DECOMPRESS (FORMAT)
        var decompress_function
        var n, index

        n = ceiling(log2(k-1))
        pop (uncompressed_data, n, index)

        if (compressor_state <> "CO") then
          # get-method-list in decode-indicator-flags has to assume
          # one of the methods from FORMAT to get a list of methods
          # used.  This may not be the one actually used according to
          # the index so the list of methods (ie method_chosen) must
          # be reparsed from the original main_list_elt in light of
          # the updated information.

          format-get-method-list (main_list_elt, method_chosen,
                                  new_method (value (index)))
          current_set = current_set * k + value (index)
        endif

        decompress_function = lookup-decompress-function (
                        extract-name (new_method(value (index))))

        call decompress_function (
                        extract-name (new_method(value (index))))

    end DECOMPRESS


    BUILD (FORMAT, 100%, format)
        var j, i
        var build_function
        var temp_format
```



```
        if (compressor_state = "CO") then
```

```
        j = current_set mod k


        current_set = current_set / k
        build_function = lookup-build-function (
                                extract-name (new_method(j)))
        call build_function (extract-name (new_method(j)),
                                100%, format)
      else
        for i = 0 to (k - 1)
          empty (temp_format)
          build_function = lookup-build-function (
                                extract-name (new_method(i)))
          call build_function (extract-name (new_method(i),
                                100%, temp_format)

          foreach item in temp_format
            append (item.id, new_method(i))
            append (format, item)
          end loop
        end loop

        DISCARD (format)

      endif
    end BUILD
```

A.4.16 CRC

```
CRC(n,P%)

COMPRESS (CRC)
    var crc_function

    crc_function = lookup-crc-function (n)
    call crc_function (n,  crc_static + crc_dynamic, crc)
    push (compression_stack, crc)
    return true
end COMPRESS


DECOMPRESS (CRC)
    pop (compression_stack, n, crc)
end DECOMPRESS


BUILD (CRC, P%, format)
    var item

    item.P = P
    item.N = n
    empty (item.id)
    append (format, item)
end BUILD
```

A.4.16.1 MSN-LSB

```
MSN-LSB(k,p,P%)

COMPRESS (MSN-LSB)
    var context_val, item, temp, lsb_val, p_item
    var n, i



    context (enc_index, 1, context_val)
    n = length (context_val)
    p_item = str (n, p)
```

```
      for i = 1 to r
        context (enc_index, i, context_val)
```



```
        # check MSN in interval
        # [value (context_val)-p, value (context_val)-p+2^k]

        temp = MSN - context_val + p_item
        if ((value (temp) < 0) or (value (temp) >= 2^k)) then
          return false
        endif
      end loop

      lsb_val = lsb (MSN, k)
      push (compression_stack, lsb_val)
      save (enc_index, MSN, storage)
      enc_index = enc_index + 1
      msn_bits = k
      return true
  end COMPRESS


  DECOMPRESS (MSN-LSB)
      var context_val, interval_start, interval_end, recd, p_item
      var new_item, twok_item, temp, lsbs, twok_extra
      var n, m, new, start, end

      context (enc_index, 1, context_val)
      n = length (context_val)
      p_item = str (n, p)
      twok_item = str (n, 2^k)
      twok_extra = str (n, 2^(k + length (msn_lsbs)))

      pop (compression_stack, k, temp)
      recd = concat (msn_lsbs, temp)

      interval_start = context_val - p_item
      interval_end = interval_start + twok_extra
      new_item = concat (msb (interval_start, (n-k-length (msn_lsbs))),
                         recd)

      # check whether value (new_item) is in interval
      # [value (interval_start), value (interval_end)]
```

```
      # allowing for the interval to wrap over zero. If not then
      # recalculate new_item

      start = value (interval_start)
      end = value (interval_end)
      new = value (new_item)

      if (((start < end) and ((new < start) or (new > end))) or
```



```
          ((start > end) and ((new > start) or (new < end)))) then

        new_item = concat (msb (interval_end, (n-k-length (msn_lsbs))),
                           recd)
      endif

      MSN = new_item
      save (enc_index, new_item, storage)
      enc_index = enc_index + 1
   end DECOMPRESS


   BUILD (MSN-LSB, P, format)
      var item

      item.P = P
      item.N = k
      empty (item.id)
      append (format, item)
   end BUILD
```

A.4.16.2 MSN-IRREGULAR

```
   MSN-IRREGULAR(n,P%)

   COMPRESS (MSN-IRREGULAR)
       push (compression_stack, MSN)
       save (enc_index, MSN, storage)
       enc_index = enc_index + 1
       msn_bits = n
       return true
   end COMPRESS


   DECOMPRESS (MSN-IRREGULAR)
       pop (compression_stack, n, MSN)
       save (enc_index, MSN, storage)
       enc_index = enc_index + 1
   end DECOMPRESS


   BUILD (MSN-IRREGULAR, P, format)
      var item

      item.P = P
```

```
      item.N = n
      empty (item.id)
      append (format, item)
   end BUILD
```


A.4.16.3 SET-MSN

```
   SET-MSN(n)

   COMPRESS (SET-MSN)
       var item

       pop (uncompressed_data, n, item)
       MSN = item
       return true
   end COMPRESS


   DECOMPRESS (SET-MSN)
       push (uncompressed_data, MSN)
   end DECOMPRESS
```

```
BUILD (SET-MSN, 100%, format)
end BUILD
```

A.5 ABNF description of the input language

The following is an ABNF description of a ROHC [1] profile generated using EPIC-LITE:

```
<profile>                   =   <profile_identifier> <ws>
                                <max_formats> <ws>
                                <max_sets> <ws>
                                <bit_alignment> <ws>
                                <npatterns> <ws>
                                <CO_packet>
                                [<ws> <IR_DYN_packet>]
                                [<ws> <IR_packet>]

<ws>                        =   1*(%x09 | %x0A | %x0D | %x20)
                                    ; white space used as
                                    ; delimiters

<profile_identifier>        =   "profile_identifier" <ws>
                                <hex_integer>
```



```
<max_formats>               =   "max_formats" <ws> <integer>

<max_sets>                  =   "max_sets" <ws> <integer>

<bit_alignment>             =   "bit_alignment" <ws> <integer>

<npatterns>                 =   "npatterns" <ws> <integer>
```

```
   <CO_packet>                  =       "CO packet" <ws> <encoding_name>

   <IR_DYN_packet>              =       "IR-DYN packet" <ws>
                                        <encoding_name>

   <IR_packet>                  =       "IR packet" <ws> <encoding_name>

   <integer>                    =       1*(<digit>)

   <digit>                      =       "0" | "1" | "2" | "3" | "4" |
                                        "5" | "6" | "7" | "8" | "9"

   <hex_integer>                =       "0x" <hex_digit> *(<hex_digit>)

   <hex_digit>                  =       <digit> | "a" | "b" | "c" | "d"
                                        | "e" | "f" | "A" | "B" | "C" |
                                        "D" | "E" | "F"
```

The following is an ABNF description of a new encoding method written using the input language (note that the previous ABNF rules still apply). Comments are contained between a ";" symbol and the end of the line, and are ignored in the input language.

```
   <encoding_method>            =       <encoding_name> <ws> "="
                                        1*(<ws> <field_encoding>)
```



```
   <field_encoding>             =       <encoding_name> <ws>
                                        *(<ws> "|" <ws> <encoding_name>)
```

```
<encoding_name>             =       <name> ["(" <parameter> *("," <parameter>) ")"]

<name>                      =       <letter> *(<letter> | <digit> | "_" | "-" | "/" | ".")

<letter>                    =       "A" | "B" | "C" | ... | "X" | "Y" | "Z" | "a" | ... | "z"

<parameter>                 =       <value> | <length> | <offset> | <probability> | <encoding_name>

<probability>               =       <digit> [<digit>] [<digit>] ["." <digit> [<digit>]] "%"

<value>                     =       <integer> | <hex_integer> | <binary_integer>

<binary_integer>            =       "0b" <bit> *(<bit>)

<bit>                       =       "0" | "1"

<length>                    =       <integer>

<offset>                    =       ["-"] <integer>
```

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.