Framework for EPIC-LITE
<draft-ietf-rohc-epic-lite-00.txt>

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of [RFC-2026].

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

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

Abstract

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

The RObust Header Compression [ROHC] 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] profiles.
1. Introduction

This document describes a plug-in extension for the [ROHC] 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] 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].

Profile

A [ROHC] 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:

```
+---+---+---+---+--------------------+--------------------+---+
| 0 | 1 | 1 | 0 | Compressed Field 1 | Compressed Field 2 |...
+---+---+---+---+--------------------+--------------------+---+
\ \ / / \ \ / /
\ \ / / \ /
Indicator Flags Compressed Fields
```

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] 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.

\[
\begin{array}{c}
| \text{Uncompressed packet} \quad \text{Uncompressed packet} |
\end{array}
\]

\[
\begin{array}{c}
| \text{Selecting the context} \quad \text{Verifying correct decompression} |
\end{array}
\]

\[
\begin{array}{c}
| \text{Running the state machine} \quad \text{Decompressing the fields} |
\end{array}
\]

\[
\begin{array}{c}
| \text{Set of header formats} \quad \text{Chosen format} |
\end{array}
\]

\[
\begin{array}{c}
| \text{Compressing the fields} \quad \text{Reading the indicator flags} |
\end{array}
\]

\[
\begin{array}{c}
| \text{Compressed header} \quad \text{Decapsulating from ROHC packet} |
\end{array}
\]

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] 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] 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] 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.

```
                             +-----------------------+
                             |    Input Stage 1:     |
                             |     Resolve input     |
                             | into set of compressed|
                             |    header formats     |
                             |    (Section A.3.1)    |
                             +-----------------------+
                                 |
                                 v
                             +-----------------------+
                             |    Input Stage 2:     |
                             | Run Huffman algorithm |
                             |   to generate flags   |
                             |    (Section A.3.2)    |
                             +-----------------------+
```

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 [HUFF] 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] 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]. For example, in ABNF the syntax for defining a new encoding method in terms of existing ones is as follows:

\[
\begin{align*}
<\text{encoding\_method}\rangle & = \quad <\text{encoding\_name}\rangle \ <\text{ws}\> \ "=" \\
& \quad 1\*(<\text{ws}\> \ <\text{field\_encoding}\rangle)
\end{align*}
\]

\[
\begin{align*}
<\text{field\_encoding}\rangle & = \quad <\text{encoding\_name}\rangle \\
& \quad *(<\text{ws}\> \ "|" \ <\text{ws}\> \ <\text{encoding\_name}\rangle)
\end{align*}
\]
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 = Version
              Traffic_Class
              ECT_Flag
              CE_Flag
              Flow_Label
              Payload_Length
              Next_Header
              Hop_Limit
              Source_Address
              Destination_Address

Version = STATIC-KNOWN(4,6)

Traffic_Class = C(STATIC(99.9%)) | IRREGULAR(6,0.1%)

ECT_Flag = C(STATIC(99.9%)) | IRREGULAR(1,0.1%)

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

Flow_Label = STATIC-UNKNOWN(20)

Payload_Length = INFERRED-SIZE(16,288)

Next_Header = STACK-TO-CONTROL(8)

Hop_Limit = C(STATIC(99%)) | IRREGULAR(8,1%)

Source_Address = STATIC-UNKNOWN(128)

Destination_Address = STATIC-UNKNOWN(128)

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. STATIC-KNOWN) 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] 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]. 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] 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]. 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.

The STATIC encoding has two variants that are useful for packet classification:

5.1.1. STATIC-KNOWN

ABNF notation: "STATIC-KNOWN(" <length> "," <value> ")"

The STATIC-KNOWN encoding method is a special version of STATIC encoding that can be used when a field always takes one well-known value. The length and integer value of the field are given as parameters after the encoding.

The STATIC-KNOWN command does not need a probability parameter as it is automatically used to compress the field with 100% certainty. If the STATIC-KNOWN encoding method is assigned to a field then the field SHOULD NOT be assigned any other encoding methods within the same field encoding (if this rule is broken then the compressor and
decompressor will still function correctly, but the compression efficiency will be very low). Note that this is true for all fields that are not given probability parameters.

Since the STATIC-KNOWN fields take only one constant value for any packet, the compressor MAY wish to use them for profile selection. It is not possible to compress a certain header using a certain ROHC profile unless the STATIC-KNOWN fields in the header take the values specified in the profile. Conversely, if every STATIC-KNOWN field in the header takes the specified value then it is likely that the ROHC profile can be used to successfully compress the header. More detail on profile selection can be found in Appendix A.1.1.

5.1.2. STATIC-UNKNOWN

ABNF notation: "STATIC-UNKNOWN(" <length> ")"

The STATIC-UNKNOWN encoding method is a special version of STATIC encoding that can be used when a field always takes one value. Unlike the STATIC-KNOWN encoding method this value is not known a-priori and must be transmitted to the decompressor using an IR packet. For this reason, only the length of the field (not its value) is given as a parameter.

Since the STATIC-UNKNOWN fields indicate the flow to which a packet belongs, the compressor MAY wish to use them for context selection. It is not possible to compress a certain header using a certain ROHC context unless the STATIC-UNKNOWN fields in the header take the values specified in the context. Conversely, if every STATIC-UNKNOWN field in the context takes the specified value then it is likely that the ROHC context can be used to successfully compress the header. More detail on context selection can be found in Appendix A.1.1.

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.

Note that the VALUE encoding method is similar to STATICKNOWN encoding except that the field does not have to be compressed using VALUE encoding for 100% of the time. Consequently an additional probability parameter is included to give the percentage of time for which the field is expected to be compressed using VALUE encoding.

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].

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 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.

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:

\[
\text{INFERRED-TRANSLATE}(8,16,41,0x86DD,4,0x0800) \quad ; \text{GRE Protocol}
\]

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):

\[ L = 8 \times v + p \]

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) ; Add MSN
INFERRED-OFFSET(32) ; Sequence Number
STACK-POP-MSN(32) ; Remove MSN
C(STATIC(99%)) | ; Sequence Number offset
  LSB(8,-1,0.7%) |
  C(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-SCALED

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

The INFERRED-SCALED encoding method compresses a field that usually increases by a fixed value for consecutive headers.

Note that this encoding is a more powerful version of INFERRED-OFFSET encoding, with two additional variables called scale and NBO. Each time the base field increases by 1, the field marked as INFERRED-SCALED increases by the value contained in scale (for INFERRED-OFFSET the scale factor is always 1). Additionally, NBO specifies whether the Network Byte Order of the field should be reversed before the scale factor is added.

The precise values of scale and NBO do not affect interoperability (since there is always a suitable value for the offset bits given any choice of scale and NBO), and so the decision on when to change the scale factor and the Network Byte Order is implementation-specific.

Once the scale, NBO and offset bits have been determined, they are used to replace the original field in the uncompressed header (and subsequently compressed by the next encoding methods encountered). For example the TCP Sequence Number can be compressed as follows:

```
STACK-PUSH-MSN(32) ; Add MSN
INFERRED-SCALED(32) ; Sequence Number
STACK-POP-MSN(32) ; Remove MSN
C(IRREGULAR-PADDED(32,0,99%)) | ; Sequence Number Scale
  C(IRREGULAR-PADDED(32,16,0.9%)) |
  IRREGULAR-PADDED(32,32,0.1%)
VALUE(1,0,100%) ; Sequence Number NBO
C(STATIC(77%)) | ; Sequence Number Offset
  C(LSB(11,-1,22%)) |
  C(LSB(16,-2049,0.9%)) |
  IRREGULAR(32,0.1%)
```

5.8. 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:
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).

5.9. 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.

5.10. 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] 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%) |
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".

5.11. 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
C(STATIC(50%)) |  
D(IRREGULAR(8,50%))  
C(STATIC(75%)) |  
IRREGULAR(4,25%)  
```

Price et al.
5.11.1. 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

C(STATIC(50%)) | ; Header_Chain order
D(IRREGULAR(8,50%))

C(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.

5.12. 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).

5.12.1. C flag

ABNF notation: "C(" <encoding_name> ")"
The C flag is used to make encoding methods available to a CO packet only. In the IR and IR-DYN packets, any encoding method marked with a C flag is ignored.

An example of the C flag in action is given below:

```
C(STATIC(99%)) |                   ; Hop Limit
IRREGULAR(8,1%)
```

In the CO packets the Hop Limit field has a 99% probability of remaining static and a 1% probability of changing. However, in the IR and IR-DYN packets it is treated as IRREGULAR and always transmitted in full.

5.12.2. D flag

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

The D flag is used to make encoding methods available to an IR(-DYN) packet only. In the CO packets, any encoding method marked with a D flag is ignored.

5.12.3. 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:

```
C(STATIC(99%)) |                   ; Window
C(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. This reflects the behavior of the TCP Window, which usually takes a constant value (the maximum window size) but occasionally takes a value slightly lower than this when packets are queued at the receiver.

5.13. 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:

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

5.14. 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 encoded as STATICKNOWN or STATICUNKNOWN are placed in crc_static, and the remaining fields 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: \[ C(x) = 1 + x^2 + x^{15} + x^{16} \]

5.15. 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 MSN encoding methods:

5.15.1. 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.

5.15.2. 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.

6. Creating a new ROHC profile

This chapter describes how to generate new [ROHC] 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] 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], it is important for EPIC-LITE not to use the bit patterns 111XXXXXX 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] 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] 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].

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] have been important sources of ideas and knowledge. Basic Huffman encoding [HUFF] was enhanced for the specific tasks of robust, efficient header compression.

Thanks to

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

for valuable input and review.
9. References

[ROHC] "RObust Header Compression (ROHC)", Carsten Bormann et al, RFC3095, Internet Engineering Task Force, July 2001


[RFC-2119] "Key words for use in RFCs to Indicate Requirement Levels", Scott Bradner, Internet Engineering Task Force, March 1997


10. Authors’ addresses

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

Robert Hancock
Email: robert.hancock@roke.co.uk
Tel: +44 1794 833601

Stephen McCann
Email: stephen.mccann@roke.co.uk
Tel: +44 1794 833341

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

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

Paul Ollis
Email: paul.ollis@roke.co.uk
Tel: +44 1794 833168

Roke Manor Research Ltd
Romsey, Hants, SO51 0ZN
United Kingdom
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] 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]).
- **npatterns**: Number of bit patterns available for EPIC-LITE in the first word of the compressed header (set to 224 for compatibility with [ROHC]).
- **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
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

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 (val, k)** Returns the least significant k bits of a value (var) in a width defined value
- **msb (val, k)** Returns the most significant k bits of a 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.

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 and is incremented and decremented.
throughout compression and decompression. Some encoding methods don’t have any context associated with them, for example, STATIC-KNOWN. 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 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

There should be a one-to-one mapping of some kind between the list method_chosen and the list of flags generated by the BUILD-FLAGS procedure (see A.3.2) with functions to perform this mapping:

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-indicator-flags (method_chosen, flags_list_elt)  
  Finds flags_list_elt from method_chosen

get-method-list (flags_list_elt, method_chosen, encoding)  
  Finds method_chosen from flags_list_elt

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

Other constructs:

foreach item in list:
  end loop

  provides for iteration over all the elements of a list

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

Price et al.
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

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)

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 STATIC-KNOWN and the STATIC-UNKNOWN fields 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.

Note that it 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...

Price et al.
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

discard

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:

procedure INDICATOR-FLAGS

    var n, bit, temp
    var extra_bits

    get-indicator-flags (method_chosen, flags_list_elt)

    push (compressed_data, flags_list_elt.flaglength, flags_list_elt.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 (value(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]. 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 READ-FLAGS
  var found, n, size, bit, k, len
  var temp, flags

  found = 0

  for each item in flags_list
    if (found = 0)
      if (value (top (received_data, flags_list_elt.flaglength)) =
        flags_list_elt.flags) then
        found = 1
        pop (received_data, flags_list_elt.flaglength, flags)
        get-method-list (flags_list_elt, method_chosen)
        len = flags_list_elt.flaglength
        n = get-header-length (method_chosen)

    endif

  endif

end loop
# 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)

eend 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.

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 COMRESS 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 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:

---

```plaintext
# # 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
```

---

```plaintext
# # 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 list
    if ((count >= max_formats) and (last_item.P <> item.P)) then
        break
    endif

    append (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...

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_reserv is set

procedure BUILD-FLAGS (main_list, do_reserve)

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

natural-sort (main_list, COMPARE)

DISCARD (main_list, max_formats)

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

for i = 0 to (max_formats - 2)

    if (i = (max_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
w = head-of (main_list)
for i = 0 to (max_formats - 1)
z = w
w.flaglength = 0
do
if (z = reserve [0]) then
w.flaglength += 1
endif
if (flag and z = reserve [1]) then
w.flaglength += 1
endif
w.flaglength += 1
z = z.parent
loop until z.parent = NULL_ITEM
w = next-item (w)
end loop
w = tail-of (main_list)
i = max_formats - 1
value = 0
for i = 0 to (max_formats - 1)
w.flags = str (w.flaglength, value)
prev_length = w.flaglength
w = previous-item (w)
value = (value + 1) * 2^(w.flaglength - prev_length)
end loop
end BUILD_FLAGS
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 a list each item of which has three parts:

- `list_item.flags` The bit pattern to be pushed on the front of a compressed header
- `list_item.flaglength` The number of flags
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

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:

<table>
<thead>
<tr>
<th>Compressed header format</th>
<th>No. of flags</th>
<th>Bit pattern of flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1010</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1100</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>11010</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>110110</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>110111</td>
</tr>
</tbody>
</table>

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] 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 (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.4.1.1.  STATIC-KNOWN(n,v)

COMPRESS (STATIC-KNOWN)
  var item

  if (value (top (uncompressed_data, n)) <> v) then
    return false
  else
    pop (uncompressed_data, n, item)
  endif
return true
endif
derived COMPRESS

DECOMPRESS (STATIC-KNOWN)
push (uncompressed_data, n, item)
derived DECOMPRESS

BUILD (STATIC-KNOWN, 100%, format)
derived BUILD

A.4.1.2. STATIC-UNKNOWN(n)

COMPRESS (STATIC-UNKNOWN)
var item, context_val
var i

if compressor_state = "IR" then
  pop (uncompressed_data, n, item)
push (compressed_data, item)
save (enc_index, item, storage)
else
  # check that the value to be compressed matches each of the r
  # values stored in context for this encoding - if not then the
  # flow has changed and cannot be compressed using this context
  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)
endif

enc_index = enc_index + 1
return true
derived COMPRESS

DECOMPRESS (STATIC-UNKNOWN)
var item

if compressor_state = "IR" then
  pop (compressed_data, n, item)
save (enc_index, item, storage)
else
  context (enc_index, 1, item)
endif
enc_index = enc_index - 1
push (uncompressed_data, item)
derived DECOMPRESS
BUILD (STATIC-UNKNOWN, 100%, format)
    var item

    item.P = 100
    if (compressor_state = "IR") then
        item.N = n
    else
        item.N = 0
    endif
    empty (item.id)
    append (format, item)
end BUILD

A.4.2. IRREGULAR(n, P%)

COMPRESS (IRREGULAR)
    var item

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

DECOMPRESS (IRREGULAR)
    var item

    pop (compressed_data, 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(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 (value (item), k)
push (compressed_data, temp)
save (enc_index, item, storage)
enc_index = enc_index + 1
return true
dendif end COMPRESS

DECOMPRESS (IRREGULAR-PADDED)
var item, temp
var full
full = 0
pop (compressed_data, k, temp)
full = full + value (temp)
item = str (n, full)
push (uncompressed_data, item)
save (enc_index, item, storage)
enc_index = enc_index - 1
dend DECOMPRESS

BUILD (IRREGULAR-PADDED, P, format)
var item
item.P = P
item.N = k
empty (item.id)
append (format, item)
dend BUILD

A.4.3. 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
dendif end COMPRESS

DECOMPRESS (VALUE)
push (uncompressed_data, n, v)
save (enc_index, v, storage)
enc_index = enc_index - 1
dend 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(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 (value (item), k)
push (compressed_data, 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 (compressed_data, k, recd)
interval_start = context_val - p
interval_end = interval_start + twok_item
new_item = concat (msb (value (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(value(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 (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.4.6.1. 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 (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 (n)

COMPRESS (STACK-PUSH-MSN)
var temp

Price et al. [PAGE 57]
\[
\text{temp} = \text{str}(n, \text{value(MSN)} \mod 2^n)
\]
\[
push(\text{control_data}, \text{temp})
\]
\[
\text{return true}
\]
end COMPRESS

DECOMPRESS (STACK-PUSH-MSN)
\[
\text{var item}
\]
\[
\text{pop(\text{control_data}, item)}
\]
end DECOMPRESS

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

A.4.6.4. STACK-POP-MSN (n)

COMPRESS (STACK-POP-MSN)
\[
\text{var item, temp}
\]
\[
\text{pop(\text{uncompressed_data}, n, item)}
\]
\[
\text{temp} = \text{str}(n, \text{value(MSN)} \mod 2^n)
\]
\[
\text{if (item} \neq \text{temp)}
\]
\[
\#\text{value on stack wasn’t MSN}
\]
\[
\text{return false}
\]
endif
\[
\text{return true}
\]
end COMPRESS

DECOMPRESS (STACK-POP-MSN)
\[
\text{var temp}
\]
\[
\text{temp} = \text{str}(n, \text{value(MSN)} \mod 2^n)
\]
\[
push(\text{uncompressed_data}, \text{temp})
\]
end DECOMPRESS

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

A.4.6.5. STACK-ROTATE (n,m)

COMPRESS (STACK-ROTATE)
\[
\text{rotate(\text{control_data}, n, m)}
\]
\[
\text{return true}
\]
end COMPRESS

DECOMPRESS (STACK-ROTATE)
\[
\text{rotate(\text{control_data}, n, (n - m))}
\]
end DECOMPRESS

BUILD (STACK-ROTATE, 100\%, format)
end BUILD
A.4.7.1. INFERRED-TRANSLATE

\[(n, m, a(0), b(0), a(1), b(1), \ldots, a(k-1), b(k-1))\]

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(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(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-SCALED(n)

COMPRESS (INFERRED-SCALED)
var item, base, offset, scale, nbo, temp

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

choose (scale_val = any n-bit value) # compressor choice
choose (nbo_val = 0 or 1) # depending whether value(item) is in
  # network byte order (nbo_val = 1) or
  # not
scale = str (n, scale_val)
nbo = str (1, nbo_val)
if nbo_val = 0 then
  temp = item
else
  temp = byte-swap (item)
endif
offset = temp - scale * base
push (uncompressed_data, offset)
push (uncompressed_data, nbo)
push (uncompressed_data, scale)
push (uncompressed_data, base)
return true
end COMPRESS

DECOMPRESS (INFERRED-SCALED)
  var item, base, offset, scale, nbo, temp
  var scale_val, nbo_val
  pop (uncompressed_data, n, base)
pop (uncompressed_data, n, scale)
pop (uncompressed_data, 1, nbo)
pop (uncompressed_data, n, offset)
  temp = offset + scale * base
  if nbo_val = 0 then
    item = temp
  else
    item = byte-swap (temp)
  endif
  push (uncompressed_data, item)
push (control_data, base)
end DECOMPRESS

BUILD (INFERRED-SCALED, 100%, format)
end BUILD

A.4.8. OPTIONAL (new_method)
COMPRESS (OPTIONAL)
var flag
var compress_function
var can_compress, n
var enc

flag = top (control_data)
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

    n = count-bits (new_method, enc)
push (compressed_data, n, 0)
store (method_chosen, OPTIONAL, enc)
can_compress = 1
endif

return can_compress
end COMPRESS

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

flag = top (control_data)
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)
    n = count-bits (new_method, enc)
pop (compressed_data, n, item)
endif

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.4.9. MANDATORY (new_method)

COMPRESS (MANDATORY)
var flag
var compress_function
var can_compress

flag = top (control_data)
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

return can_compress
end COMPRESS

DECOMPRESS (MANDATORY)
var decompress_function

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

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

A.4.10. 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 - ((k - 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.4.11. 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)]
var compress_function

scale_len = floor( value(top(control_data)) / 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
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])
            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])
            if (length (present[i]) <> 1) then
                return false
            endif
        else
            i = i + 1
        endif
    end while

# then compress using any methods which haven't been used yet

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])

        else
            i = i + 1
        endif
    end for

Price et al.
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, bits
var present [0..(k - 1)]
var presence_item, order_item

bits = ceiling (log2(k-1))

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

while (order <> 0)
  i = value (lsb (order, bits))
  order = (order ÷ i )/ 2^bits
push (control_data, present [i])
decompress_function = lookup-decompress-function
  (extract-name (new_method(i)))
call decompress_function (extract-name (new_method(i)))
pop (control_data, present[i])
end while
end DECOMPRESS

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

A.4.11.1. LIST-NEXT ($n$, new_method(0), new_method(1), ..., new_method(k-1), $v$(0), $v$(1), ..., $v$(j))

# This pseudo-code 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

  Price et al. [PAGE 67]
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

# then compress using any methods which haven't been used yet

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 and BUILD are the same as for LIST

A.4.12.1. C (new_method)

COMPRESS (C)
  var compress_function
  var can_compress

  if (compressor_state <> "CO") then
    return false
  else
    compress_function = lookup-compress-function
                        (extract-name (new_method))
    can_compress = call compress_function
                      (extract-name (new_method))
    return can_compress
  endif
end COMPRESS

DECOMPRESS (C)
  var decompress_function

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

  call decompress_function (extract-name (new_method))
end DECOMPRESS

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

  if (compressor_state = "CO") then
    build_function = lookup-build-function (new_method)
call build_function (extract-name (new_method), P, format)
endif
end BUILD

A.4.12.2. D (new_method)

COMPRESS (D)
var compress_function
var can_compress

if (compressor_state = "CO") then
  return false
else
  compress_function = lookup-compress-function
                      (extract-name (new_method))

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

  return can_compress
endif
end COMPRESS

DECOMPRESS (D)
var decompress_function

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

call decompress_function (extract-name (new_method))
end DECOMPRESS

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

if (compressor_state <> "CO") then
  build_function = lookup-build-function (new_method)

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

A.4.12.3. 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)
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)
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.4.13.  FORMAT(new_method(0), ..., new_method(k - 1))

COMPRESS (FORMAT)
var n, can_compress, i, index_val
var compress_function
var check, index

n = ceiling(log2(k-1))

if (compressor_state <> "CO") then
choose (index_val < k) # compressor choice
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 + max_sets * index_val
    max_sets = max_sets * k
    index = str (n, index_val)
else
    context (enc_index, 1, index)
for i = 2 to r
    context (enc_index, i, check)
    if (check <> index) then
        return false
    endif
end loop

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

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

if can_compress = false then
    return false
else
    push (uncompressed_data, index)
    save (enc_index, index, storage)
    enc_index = enc_index + 1
endif
return true
end COMPRESS

DECOMPRESS (FORMAT)
var decompress_function
var n, index

n = ceiling(log2(k-1))
if (compressor_state <> "CO") then
    pop (uncompressed_data, n, index)
    current_set = current_set * k + value (index)
else
    context (enc_index, 1, index)
endif

save (enc_index, index, storage)
enc_index = enc_index - 1
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

Price et al.
current_set = floor(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.4.14. CRC(n, P%)

COMPRESS (CRC)
var crc_function

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

DECOMPRESS (CRC)
    pop (compressed_data, 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.4.15.1. MSN-LSB (k, p, P%)

COMPRESS (MSN-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 = MSN

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

  lsb_val = lsb (value (item), k)
push (compressed_data, lsb_val)
save (enc_index, item, 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 + msn_bits))

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

interval_start = context_val - p_item
interval_end = interval_start + twok_extra
new_item = concat (msb (value(interval_start), (n-k-msn_bits)),
                    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 (value (interval_end), (n-k-msn_bits)), 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.4.15.2. MSN-IRREGULAR (n, P%)

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

DECOMPRESS (MSN-IRREGULAR)
pop (compressed_data, 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.5. ABNF description of the input language

The following is an ABNF description of a [ROHC] 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>)
```

Price et al.                                                 [PAGE 76]
<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>]] "."<percentage>
<value> = <integer> | <hex_integer> | <binary_integer>
<binary_integer> = "0b" <bit> *(<bit>)
<bit> = "0" | "1"
<length> = <integer>
[offset] = [":"] <integer>