            Formal Notation for RObust Header Compression (ROHC-FN)

Status of This Memo

Copyright Notice

Abstract

   This document defines Robust Header Compression - Formal Notation
   (ROHC-FN), a formal notation to specify field encodings for
   compressed formats when defining new profiles within the ROHC
   framework. ROHC-FN offers a library of encoding methods that are
   often used in ROHC profiles and can thereby help to simplify future
   profile development work.

Table of Contents

1. Introduction

   Robust Header Compression - Formal Notation (ROHC-FN) is a formal
   notation designed to help with the definition of ROHC [RFC4995]
   header compression profiles. Previous header compression profiles
   have been so far specified using a combination of English text
   together with ASCII Box notation. Unfortunately, this was sometimes
   unclear and ambiguous, revealing the limitations of defining complex
   structures and encodings for compressed formats this way. The
   primary objective of the Formal Notation is to provide a more
   rigorous means to define header formats -- compressed and
   uncompressed -- as well as the relationships between them. No other
   formal notation exists that meets these requirements, so ROHC-FN aims
   to meet them.

   In addition, ROHC-FN offers a library of encoding methods that are
   often used in ROHC profiles, so that the specification of new
   profiles using the formal notation can be achieved without having to
   redefine this library from scratch. Informally, an encoding method
   defines a two-way mapping between uncompressed data and compressed
   data.

2. Terminology

   o Compressed format

      A compressed format consists of a list of fields that provides
      bindings between encodings and the fields it compresses. One or
      more compressed formats can be combined to represent an entire
      compressed header format.

   o Context

      Context is information about the current (de)compression state of
      the flow. Specifically, a context for a specific field can be
      either uninitialised, or it can include a set of one or more
      values for the field's attributes defined by the compression
      algorithm, where a value may come from the field's attributes
      corresponding to a previous packet. See also a more generalized
      definition in Section 2.2 of [RFC4995].

   o Control field

      Control fields are transmitted from a ROHC compressor to a ROHC
      decompressor, but are not part of the uncompressed header itself.

o Encoding method, encodings

   Encoding methods are two-way relations that can be applied to
   compress and decompress fields of a protocol header.

o Field

   The protocol header is divided into a set of contiguous bit
   patterns known as fields. Each field is defined by a collection
   of attributes that indicate its value and length in bits for both
   the compressed and uncompressed headers. The way the header is
   divided into fields is specific to the definition of a profile,
   and it is not necessary for the field divisions to be identical to
   the ones given by the specification(s) for the protocol header
   being compressed.

o Library of encoding methods

   The library of encoding methods contains a number of commonly used
   encoding methods for compressing header fields.

o Profile

   A ROHC [RFC4995] profile is a description of how to compress a
   certain protocol stack. Each profile consists of a set of formats
   (for example, uncompressed and compressed formats) along with a
   set of rules that control compressor and decompressor behaviour.

o ROHC-FN specification

   The specification of the set of formats of a ROHC profile using
   ROHC-FN.

o Uncompressed format

   An uncompressed format consists of a list of fields that provides
   the order of the fields to be compressed for a contiguous set of
   bits whose bit layout corresponds to the protocol header being
   compressed.

3. Overview of ROHC-FN

   This section gives an overview of ROHC-FN. It also explains how
   ROHC-FN can be used to specify the compression of header fields as
   part of a ROHC profile.

3.1. Scope of the Formal Notation

   This section explains how the formal notation relates to the ROHC
   framework and to specifications of ROHC profiles.

   The ROHC framework [RFC4995] provides the general principles for
   performing robust header compression. It defines the concept of a
   profile, which makes ROHC a general platform for different
   compression schemes. It sets link layer requirements, and in
   particular negotiation requirements, for all ROHC profiles. It
   defines a set of common functions such as Context Identifiers (CIDs),
   padding, and segmentation. It also defines common formats (IR, IR-
   DYN, Feedback, Add-CID, etc.), and finally it defines a generic,
   profile independent, feedback mechanism.

   A ROHC profile is a description of how to compress a certain protocol
   stack. For example, ROHC profiles are available for RTP/UDP/IP and
   many other protocol stacks.

   At a high level, each ROHC profile consists of a set of formats
   (defining the bits to be transmitted) along with a set of rules that
   control compressor and decompressor behaviour. The purpose of the
   formats is to define how to compress and decompress headers. The
   formats define one or more compressed versions of each uncompressed
   header, and simultaneously define the inverse: how to relate a
   compressed header back to the original uncompressed header.

   The set of formats will typically define compression of headers
   relative to a context of field values from previous headers in a
   flow, improving the overall compression by taking into account
   redundancies between headers of successive packets. Therefore, in
   addition to defining the formats, a profile has to:

   o specify how to manage the context for both the compressor and the
      decompressor,

   o define when and what to send in feedback messages, if any, from
      decompressor to compressor,

   o outline compression principles to make the profile robust against
      bit errors and dropped packets.

   All this is needed to ensure that the compressor and decompressor
   contexts are kept consistent with each other, while still
   facilitating the best possible compression performance.

   The ROHC-FN is designed to help in the specification of compressed
   formats that, when put together based on the profile definition, make

up the formats used in a ROHC profile. It offers a library of
encoding methods for compressing fields, and a mechanism for
combining these encoding methods to create compressed formats
tailored to a specific protocol stack.

The scope of ROHC-FN is limited to specifying the relationship
between the compressed and uncompressed formats. To form a complete
profile specification, the control logic for the profile behaviour
needs to be defined by other means.

3.2. Fundamentals of the Formal Notation

There are two fundamental elements to the formal notation:

1. Fields and their encodings, which define the mapping between a
    header's uncompressed and compressed forms.

2. Encoding methods, which define the way headers are broken down
    into fields. Encoding methods define lists of uncompressed
    fields and the lists of compressed fields they map onto.

These two fundamental elements are at the core of the notation and
are outlined below.

3.2.1. Fields and Encodings

Headers are made up of fields. For example, version number, header
length, and sequence number are all fields used in real protocols.

Fields have attributes. Attributes describe various things about the
field. For example:

  field.ULENGTH

The above indicates the uncompressed length of the field. A field is
said to have a value attribute, i.e., a compressed value or an
uncompressed value, if the corresponding length attribute is greater
than zero. See Section 4.4 for more details on field attributes.

The relationship between the compressed and uncompressed attributes
of a field are specified with encoding methods, using the following
notation:

  field   =:=   encoding_method;

In the field definition above, the symbol "=:=" means "is encoded
by". This field definition does not represent an assignment
operation from the right hand side to the left side. Instead, it is

a two-way mapping between the compressed and uncompressed attributes
of the field. It both represents the compression and the
decompression operation in a single field definition, through a
process of two-way matching.

Two-way matching is a binary operation that attempts to make the
operands (i.e., the compressed and uncompressed attributes) match.
This is similar to the unification process in logic. The operands
represent one unspecified data object and one specified object.
Values can be matched from either operand.

During compression, the uncompressed attributes of the field are
already defined. The given encoding matches the compressed
attributes against them. During decompression, the compressed
attributes of the field are already defined, so the uncompressed
attributes are matched to the compressed attributes using the given
encoding method. Thus, both compression and decompression are
defined by a single field definition.

Therefore, an encoding method (including any parameters specified)
creates a reversible binding between the attributes of a field. At
the compressor, a format can be used if a set of bindings that is
successful for all the attributes in all its fields can be found. At
the decompressor, the operation is reversed using the same bindings
and the attributes in each field are filled according to the
specified bindings; decoding fails if the binding for an attribute
fails.

For example, the "static" encoding method creates a binding between
the attribute corresponding to the uncompressed value of the field
and the corresponding value of the field in the context.

o For the compressor, the "static" binding is successful when both
   the context value and the uncompressed value are the same. If the
   two values differ then the binding fails.

o For the decompressor, the "static" binding succeeds only if a
   valid context entry containing the value of the uncompressed field
   exists. Otherwise, the binding will fail.

Both the compressed and uncompressed forms of each field are
represented as a string of bits; the most significant bit first, of
the length specified by the length attribute. The bit string is the
binary representation of the value attribute of the field, modulo
"2^length", where "length" is the length attribute of the field.
However, this is only the representation of the bits exchanged
between the compressor and the decompressor, designed to allow

   maximum compression efficiency. The FN itself uses the full range of
   integers. See Section 4.4.2 for further details.

3.2.2. Formats and Encoding Methods

   The ROHC-FN provides a library of commonly used encoding methods.
   Encoding methods can be defined using plain English, or using a
   formal definition consisting of, for example, a collection of
   expressions (Section 4.7) and "ENFORCE" statements (Section 4.9).

   ROHC-FN also provides mechanisms for combining fields and their
   encoding methods into higher level encoding methods following a well-
   defined structure. This is similar to the definition of functions
   and procedures in an ordinary programming language. It allows
   complexity to be handled by being broken down into manageable parts.
   New encoding methods are defined at the top level of a profile.
   These can then be used in the definition of other higher level
   encoding methods, and so on.

```
   new_encoding_method           // This block is an encoding method
   {
     UNCOMPRESSED {              // This block is an uncompressed format
       field_1   [ 16 ];
       field_2   [ 32 ];
       field_3   [ 48 ];
     }

     CONTROL {                   // This block defines control fields
       ctrl_field_1;
       ctrl_field_2;
     }

     DEFAULT {                   // This block defines default encodings
                                 // for specified fields
       ctrl_field_2 =:= encoding_method_2;
       field_1      =:= encoding_method_1;
     }

     COMPRESSED format_0 {       // This block is a compressed format
       field_1;
       field_2      =:= encoding_method_2;
       field_3      =:= encoding_method_3;
       ctrl_field_1 =:= encoding_method_4;
       ctrl_field_2;
     }
```

```
  COMPRESSED format_1 {      // This block is a compressed format
    field_1;
    field_2      =:= encoding_method_3;
    field_3      =:= encoding_method_4;
    ctrl_field_2 =:= encoding_method_5;
    ctrl_field_3 =:= encoding_method_6; // This is a control field
                                        // with no uncompressed value
  }
}
```

In the example above, the encoding method being defined is called
"new_encoding_method". The section headed "UNCOMPRESSED" indicates
the order of fields in the uncompressed header, i.e., the
uncompressed header format. The number of bits in each of the fields
is indicated in square brackets. After this is another section,
"CONTROL", which defines two control fields. Following this is the
"DEFAULT" section which defines default encoding methods for two of
the fields (see below). Finally, two alternative compressed formats
follow, each defined in sections headed "COMPRESSED". The fields
that occur in the compressed formats are either:

o fields that occur in the uncompressed format; or

o control fields that have an uncompressed value and that occur in
   the CONTROL section; or

o control fields that do not have an uncompressed value and thus are
   defined as part of the compressed format.

Central to each of these formats is a "field list", which defines the
fields contained in the format and also the order that those fields
appear in that format. For the "DEFAULT" and "CONTROL" sections, the
field order is not significant.

In addition to specifying field order, the field list may also
specify bindings for any or all of the fields it contains. Fields
that have no bindings defined for them are bound using the default
bindings specified in the "DEFAULT" section (see Section 4.12.1.5).

Fields from the compressed format have the same name as they do in
the uncompressed format. If there are any fields that are present
exclusively in the compressed format, but that do have an
uncompressed value, they must be declared in the "CONTROL" section of
the definition of the encoding method (see Section 4.12.1.3 for more
details on defining control fields).

Fields that have no uncompressed value do not appear in an
"UNCOMPRESSED" field list and do not have to appear in the "CONTROL"

   field list either. Instead, they are only declared in the compressed
   field lists where they are used.

   In the example above, all the fields that appear in the compressed
   format are also found in the uncompressed format, or the control
   field list, except for ctrl_field_3; this is possible because
   ctrl_field_3 has no "uncompressed" value at all. Fields such as a
   checksum on the compressed information fall into this category.

3.3. Example Using IPv4

   This section gives an overview of how the notation is used by means
   of an example. The example will develop the formal notation for an
   encoding method capable of compressing a single, well-known header:
   the IPv4 header [RFC791].

   The first step is to specify the overall structure of the IPv4
   header. To do this, we use an encoding method that we will call
   "ipv4_header". More details on definitions of encoding methods can
   be found in Section 4.12. This is notated as follows:

     ipv4_header
     {

   The fragment of notation above declares the encoding method
   "ipv4_header", the definition follows the opening brace (see
   Section 4.12).

   Definitions within the pair of braces are local to "ipv4_header".
   This scoping mechanism helps to clarify which fields belong to which
   formats; it is also useful when compressing complex protocol stacks
   with several headers, often with the same field names occurring in
   multiple headers (see Section 4.2).

   The next step is to specify the fields contained in the uncompressed
   IPv4 header to represent the uncompressed format for which the
   encoding method will define one or more compressed formats. This is
   accomplished using ROHC-FN as follows:

```
      UNCOMPRESSED {
        version         [  4 ];
        header_length   [  4 ];
        dscp            [  6 ];
        ecn             [  2 ];
        length          [ 16 ];
        id              [ 16 ];
        reserved        [  1 ];
        dont_frag       [  1 ];
        more_fragments  [  1 ];
        offset          [ 13 ];
        ttl             [  8 ];
        protocol        [  8 ];
        checksum        [ 16 ];
        src_addr        [ 32 ];
        dest_addr       [ 32 ];
      }
```

The width of each field is indicated in square brackets. This part
of the notation is used in the example for illustration to help the
reader's understanding. However, indicating the field lengths in
this way is optional since the width of each field can also normally
be derived from the encoding that is used to compress/decompress it
for a specific format. This part of the notation is formally defined
in Section 4.10.

The next step is to specify the compressed format. This includes the
encodings for each field that map between the compressed and
uncompressed forms of the field. In the example, these encoding
methods are mainly taken from the ROHC-FN library (see Section 4.11).
Since the intention here is to illustrate the use of the notation,
rather than to describe the optimum method of compressing IPv4
headers, this example uses only three encoding methods.

The "uncompressed_value" encoding method (defined in Section 4.11.1)
can compress any field whose uncompressed length and value are fixed,
or can be calculated using an expression. No compressed bits need to
be sent because the uncompressed field can be reconstructed using its
known size and value. The "uncompressed_value" encoding method is
used to compress five fields in the IPv4 header, as described below:

```
      COMPRESSED {
        header_length  =:= uncompressed_value(4, 5);
        version        =:= uncompressed_value(4, 4);
        reserved       =:= uncompressed_value(1, 0);
        offset         =:= uncompressed_value(13, 0);
        more_fragments =:= uncompressed_value(1, 0);
```

The first parameter indicates the length of the uncompressed field in
bits, and the second parameter gives its integer value.

Note that the order of the fields in the compressed format is
independent of the order of the fields in the uncompressed format.

The "irregular" encoding method (defined in Section 4.11.3) can be
used to encode any field for which both uncompressed attributes
(ULENGTH and UVALUE) are defined, and whose ULENGTH attribute is
either fixed or can be calculated using an expression. It is a fail-
safe encoding method that can be used for such fields in the case
where no other encoding method applies. All of the bits in the
uncompressed form of the field are present in the compressed form as
well; hence this encoding does not achieve any compression.

```
        src_addr        =:= irregular(32);
        dest_addr       =:= irregular(32);
        length          =:= irregular(16);
        id              =:= irregular(16);
        ttl             =:= irregular(8);
        protocol        =:= irregular(8);
        dscp            =:= irregular(6);
        ecn             =:= irregular(2);
        dont_frag       =:= irregular(1);
```

Finally, the third encoding method is specific only to the
uncompressed format defined above for the IPv4 header,
"inferred_ip_v4_header_checksum":

```
        checksum        =:= inferred_ip_v4_header_checksum [ 0 ];
      }
    }
```

The "inferred_ip_v4_header_checksum" encoding method is different
from the other two encoding methods in that it is not defined in the
ROHC-FN library of encoding methods. Its definition could be given
either by using the formal notation as part of the profile definition
itself (see Section 4.12) or by using plain English text (see
Section 4.13).

In our example, the "inferred_ip_v4_header_checksum" is a specific
encoding method that calculates the IP checksum from the rest of the
header values. Like the "uncompressed_value" encoding method, no
compressed bits need to be sent, since the field value can be
reconstructed at the decompressor. This is notated explicitly by
specifying, in square brackets, a length of 0 for the checksum field
in the compressed format. Again, this notation is optional since the
encoding method itself would be defined as sending zero compressed

bits, however it is useful to the reader to include such notation
(see Section 4.10 for details on this part of the notation).

Finally the definition of the format is terminated with a closing
brace. At this point, the above example has defined a compressed
format that can be used to represent the entire compressed IPv4
header, and provides enough information to allow an implementation to
construct the compressed format from an uncompressed format
(compression) and vice versa (decompression).

4. Normative Definition of ROHC-FN

This section gives the normative definition of ROHC-FN. ROHC-FN is a
declarative language that is referentially transparent, with no side
effects. This means that whenever an expression is evaluated, there
are no other effects from obtaining the value of the expression; the
same expression is thus guaranteed to have the same value wherever it
appears in the notation, and it can always be interchanged with its
value in any of the formats it appears in (subject to the scope rules
of identifiers of Section 4.2).

The formal notation describes the structure of the formats and the
relationships between their uncompressed and compressed forms, rather
than describing how compression and decompression is performed.

In various places within this section, text inside angle brackets has
been used as a descriptive placeholder. The use of angle brackets in
this way is solely for the benefit of the reader of this document.
Neither the angle brackets, nor their contents form a part of the
notation.

4.1. Structure of a Specification

The specification of the compressed formats of a ROHC profile using
ROHC-FN is called a ROHC-FN specification. ROHC-FN specifications
are case sensitive and are written in the 7-bit ASCII character set
(as defined in [RFC2822]) and consist of a sequence of zero or more
constant definitions (Section 4.3), an optional global control field
list (Section 4.12.1.3) and one or more encoding method definitions
(Section 4.12).

Encoding methods can be defined using the formal notation or can be
predefined encoding methods.

Encoding methods are defined using the formal notation by giving one
or more uncompressed formats to represent the uncompressed header and
one or more compressed formats. These formats are related to each
other by "fields", each of which describes a certain part of an

uncompressed and/or a compressed header. In addition to the formats,
each encoding method may contain control fields, initial values, and
default field encodings sections. The attributes of a field are
bound by using an encoding method for it and/or by using "ENFORCE"
statements (Section 4.9) within the formats. Each of these are
terminated by a semi-colon.

Predefined encoding methods are not defined in the formal notation.
Instead they are defined by giving a short textual reference
explaining where the encoding method is defined. It is not necessary
to define the library of encoding methods contained in this document
in this way, their definition is implicit to the usage of the formal
notation.

4.2. Identifiers

In ROHC-FN, identifiers are used for any of the following:

o encoding methods

o formats

o fields

o parameters

o constants

All identifiers may be of any length and may contain any combination
of alphanumeric characters and underscores, within the restrictions
defined in this section.

All identifiers must start with an alphabetic character.

It is illegal to have two or more identifiers that differ from each
other only in capitalisation, in the same scope.

All letters in identifiers for constants must be upper case.

It is illegal to use any of the following as identifiers (including
alternative capitalisations):

o "false", "true"

o "ENFORCE", "THIS", "VARIABLE"

o "ULENGTH", "UVALUE"

o   "CLENGTH", "CVALUE"

o   "UNCOMPRESSED", "COMPRESSED", "CONTROL", "INITIAL", or "DEFAULT"

Format names cannot be referred to in the notation, although they are
considered to be identifiers. (See Section 4.12.3.1 for more details
on format names.)

All identifiers used in ROHC-FN have a "scope". The scope of an
identifier defines the parts of the specification where that
identifier applies and from which it can be referred to. If an
identifier has a "global" scope, then it applies throughout the
specification that contains it and can be referred to from anywhere
within it. If an identifier has a "local" scope, then it only
applies to the encoding method in which it is defined, it cannot be
referenced from outside the local scope of that encoding method. If
an identifier has a local scope, that identifier can therefore be
used in multiple different local scopes to refer to different items.

All instances of an identifier within its scope refer to the same
item. It is not possible to have different items referred to by a
single identifier within any given scope. For this reason, if there
is an identifier that has global scope it cannot be used separately
in a local scope, since a globally-scoped identifier is already
applicable in all local scopes.

The identifiers for each encoding method and each constant all have a
global scope. Each format and field also has an identifier. The
scope of format and field identifiers is local, with the exception of
global control fields, which have a global scope. Therefore it is
illegal for a format or field to have the same identifier as another
format or field within the same scope, or as an encoding method or a
constant (since they have global scope).

Note that although format names (see Section 4.12.3.1) are considered
to be identifiers, they are not referred to in the notation, but are
primarily for the benefit of the reader.

4.3. Constant Definitions

Constant values can be defined using the "=" operator. Identifiers
for constants must be all upper case. For example:

    SOME_CONSTANT = 3;

Constants are defined by an expression (see Section 4.7) on the
right-hand side of the "=" operator. The expression must yield a
constant value. That is, the expression must be one whose terms are

all either constants or literals and must not vary depending on the
header being compressed.

Constants have a global scope. Constants must be defined at the top
level, outside any encoding method definition. Constants are
entirely equivalent to the value they refer to, and are completely
interchangeable with that value. Unlike field attributes, which may
change from packet to packet, constants have the same value for all
packets.

4.4. Fields

Fields are the basic building blocks of a ROHC-FN specification.
Fields are the units into which headers are divided. Each field may
have two forms: a compressed form and an uncompressed form. Both
forms are represented as bits exchanged between the compressor and
the decompressor in the same way, as an unsigned string of bits; the
most significant bit first.

The properties of the compressed form of a field are defined by an
encoding method and/or "ENFORCE" statements. This entirely
characterises the relationship between the uncompressed and
compressed forms of that field. This is achieved by specifying the
relationships between the field's attributes.

The notation defines four field attributes, two for the uncompressed
form and a corresponding two for the compressed form. The attributes
available for each field are:

uncompressed attributes of a field:

o   "UVALUE" and "ULENGTH",

compressed attributes of a field:

o   "CVALUE" and "CLENGTH".

The two value attributes contain the respective numerical values of
the field, i.e., "UVALUE" gives the numerical value of the
uncompressed form of the field, and the attribute "CVALUE" gives the
numerical value of the compressed form of the field. The numerical
values are derived by interpreting the bit-string representations of
the field as bit strings; the most significant bit first.

The two length attributes indicate the length in bits of the
associated bit string; "ULENGTH" for the uncompressed form, and
"CLENGTH" for the compressed form.

Attributes are undefined unless they are bound to a value, in which
case they become defined. If two conflicting bindings are given for
a field attribute then the bindings fail along with the (combination
of) formats in which those bindings were defined.

Uncompressed attributes do not always reflect an aspect of the
uncompressed header. Some fields do not originate from the
uncompressed header, but are control fields.

4.4.1. Attribute References

Attributes of a particular field are formally referred to by using
the field's name followed by a "." and the attribute's identifier.

For example:

   rtp_seq_number.UVALUE

The above gives the uncompressed value of the rtp_seq_number field.
The primary reason for referencing attributes is for use in
expressions, which are explained in Section 4.7.

4.4.2. Representation of Field Values

Fields are represented as bit strings. The bit string is calculated
using the value attribute ("val") and the length attribute ("len").
The bit string is the binary representation of "val % (2 ^ len)".

For example, if a field's "CLENGTH" attribute was 8, and its "CVALUE"
attribute was -1, the compressed representation of the field would be
"-1 % (2 ^ 8)", which equals "-1 % 256", which equals 255, 11111111
in binary.

ROHC-FN supports the full range of integers for use in expressions
(see Section 4.7), but the representation of the formats (i.e., the
bits exchanged between the compressor and the decompressor) is in the
above form.

4.5. Grouping of Fields

Since the order of fields in a "COMPRESSED" field list
(Section 4.12.1.2) do not have to be the same as the order of fields
in an "UNCOMPRESSED" field list (Section 4.12.1.1), it is possible to
group together any number of fields that are contiguous in a
"COMPRESSED" format, to allow them all to be encoded using a single
encoding method. The group of fields is specified immediately to the
left of "=:=" in place of a single field name.

The group is notated by giving a colon-separated list of the fields
to be grouped together. For example there may be two non-contiguous
fields in an uncompressed header that are two halves of what is
effectively a single sequence number:

```
grouping_example
{
  UNCOMPRESSED {
    minor_seq_num;  // 12 bits
    other_field;    //  8 bits
    major_seq_num;  //  4 bits
  }

  COMPRESSED {
    other_field     =:= irregular(8);
    major_seq_num
    : minor_seq_num =:= lsb(3, 0);
  }
}
```

The group of fields is presented to the encoding method as a
contiguous group of bits, assembled by the concatenation of the
fields in the order they are given in the group. The most
significant bit of the combined field is the most significant bit of
the first field in the list, and the least significant bit of the
combined field is the least significant bit of the last field in the
list.

Finally, the length attributes of the combined field are equal to the
sum of the corresponding length attributes for all the fields in the
group.

4.6. "THIS"

Within the definition of an encoding method, it is possible to refer
to the field (i.e., the group of contiguous bits) the method is
encoding, using the keyword "THIS".

This is useful for gaining access to the attributes of the field
being encoded. For example it is often useful to know the total
uncompressed length of the uncompressed format that is being encoded:

```
THIS.ULENGTH
```

4.7. Expressions

   ROHC-FN includes the usual infix style of expressions, with
   parentheses "(" and ")" used for grouping. Expressions can be made
   up of any of the components described in the following subsections.

   The semantics of expressions are generally similar to the expressions
   in the ANSI-C programming language [C90]. The definitive list of
   expressions in ROHC-FN follows in the next subsections; the list
   below provides some examples of the difference between expressions in
   ANSI-C and expressions in ROHC-FN:

   o   There is no limit on the range of integers.

   o   "x ^ y" evaluates to x raised to the power of y. This has a
       precedence higher than *, / and %, but lower than unary - and is
       right to left associative.

   o   There is no comma operator.

   o   There are no "modify" operators (no assignment operators and no
       increment or decrement).

   o   There are no bitwise operators.

   Expressions may refer to any of the attributes of a field (as
   described in Section 4.4), to any defined constant (see Section 4.3)
   and also to encoding method parameters, if any are in scope (see
   Section 4.12).

   If any of the attributes, constants, or parameters used in the
   expression are undefined, the value of the expression is undefined.
   Undefined expressions cause the environment (for example, the
   compressed format) in which they are used to fail if a defined value
   is required. Defined values are required for all compressed
   attributes of fields that appear in the compressed format. Defined
   values are not required for all uncompressed attributes of fields
   which appear in the uncompressed format. It is up to the profile
   creator to define what happens to the unbound field attributes in
   this case. It should be noted that in such a case, transparency of
   the compression process will be lost; i.e., it will not be possible
   for the decompressor to reproduce the original header.

   Expressions cannot be used as encoding methods directly because they
   do not completely characterise a field. Expressions only specify a
   single value whereas a field is made up of several values: its
   attributes. For example, the following is illegal:

```
      tcp_list_length =:= (data_offset + 20) / 4;
```

   There is only enough information here to define a single attribute of
   "tcp_list_length". Although this makes no sense formally, this could
   intuitively be read as defining the "UVALUE" attribute. However,
   that would still leave the length of the uncompressed field undefined
   at the decompressor. Such usage is therefore prohibited.

4.7.1. Integer Literals

   Integers can be expressed as decimal values, binary values (prefixed
   by "0b"), or hexadecimal values (prefixed by "0x"). Negative
   integers are prefixed by a "-" sign. For example "10", "0b1010", and
   "-0x0a" are all valid integer literals, having the values 10, 10, and
   -10 respectively.

4.7.2. Integer Operators

   The following "integer" operators are available, which take integer
   arguments and return an integer result:

   o ^, for exponentiation. "x ^ y" returns the value of "x" to the
      power of "y".

   o *, / for multiplication and division. "x * y" returns the product
      of "x" and "y". "x / y" returns the quotient, rounded down to the
      next integer (the next one towards negative infinity).

   o +, - for addition and subtraction. "x + y" returns the sum of "x"
      and "y". "x - y" returns the difference.

   o % for modulo. "x % y" returns "x" modulo "y"; x - y * (x / y).

4.7.3. Boolean Literals

   The boolean literals are "false", and "true".

4.7.4. Boolean Operators

   The following "boolean" operators are available, which take boolean
   arguments and return a boolean result:

   o &&, for logical "and". Returns true if both arguments are true.
      Returns false otherwise.

   o ||, for logical "or". Returns true if at least one argument is
      true. Returns false otherwise.

   o !, for logical "not". Returns true if its argument is false.
      Returns false otherwise.

4.7.5. Comparison Operators

   The following "comparison" operators are available, which take
   integer arguments and return a boolean result:

   o ==, !=, for equality and its negative. "x == y" returns true if x
      is equal to y. Returns false otherwise. "x != y" returns true if
      x is not equal to y. Returns false otherwise.

   o <, >, for less than and greater than. "x < y" returns true if x is
      less than y. Returns false otherwise. "x > y" returns true if x
      is greater than y. Returns false otherwise.

   o >=, <=, for greater than or equal and less than or equal, the
      inverse functions of <, >. "x >= y" returns false if x is less
      than y. Returns true otherwise. "x <= y" returns false if x is
      greater than y. Returns true otherwise.

4.8. Comments

   Free English text can be inserted into a ROHC-FN specification to
   explain why something has been done a particular way, to clarify the
   intended meaning of the notation, or to elaborate on some point.

   The FN uses an end of line comment style, which makes use of the "//"
   comment marker. Any text between the "//" marker and the end of the
   line has no formal meaning. For example:

```
   //-----------------------------------------------------------------
   //     IR-REPLICATE header formats
   //-----------------------------------------------------------------

   // The following fields are included in all of the IR-REPLICATE
   // header formats:
   //
   UNCOMPRESSED {
     discriminator;    //  8 bits
     tcp_seq_number;   // 32 bits
     tcp_flags_ecn;    //  2 bits
```

   Comments do not affect the formal meaning of what is notated, but can
   be used to improve readability. Their use is optional.

   Comments may help to provide clarifications to the reader, and serve
   different purposes to implementers. Comments should thus not be

considered of lesser importance when inserting them into a ROHC-FN
specification; they should be consistent with the normative part of
the specification.

4.9. "ENFORCE" Statements

The "ENFORCE" statement provides a way to add predicates to a format,
all of which must be fulfilled for the format to succeed. An
"ENFORCE" statement shares some similarities with an encoding method.
Specifically, whereas an encoding method binds several field
attributes at once, an "ENFORCE" statement typically binds just one
of them. In fact, all the bindings that encoding methods create can
be expressed in terms of a collection of "ENFORCE" statements. Here
is an example "ENFORCE" statement which binds the "UVALUE" attribute
of a field to 5.

   ENFORCE(field.UVALUE == 5);

An "ENFORCE" statement must only be used inside a field list (see
Section 4.12). It attempts to force the expression given to be true
for the format that it belongs to.

An abbreviated form of an "ENFORCE" statement is available for
binding length attributes using "[" and "]", see Section 4.10.

Like an encoding method, an "ENFORCE" statement can only be
successfully used in a format if the binding it describes is
achievable. A format containing the example "ENFORCE" statement
above would not be usable if the field had also been bound within
that same format with "uncompressed_value" encoding, which gave it a
"UVALUE" other than 5.

An "ENFORCE" statement takes a boolean expression as a parameter. It
can be used to assert that the expression is true, in order to choose
a particular format from a list of possible formats specified in an
encoding method (see Section 4.12), or just to bind an expression as
in the example above. The general form of an "ENFORCE" statement is
therefore:

   ENFORCE(<boolean expression>);

There are three possible conditions that the expression may be in:

1. The boolean expression evaluates to false, in which case the
    local scope of the format that contains the "ENFORCE" statement
    cannot be used.

   2. The boolean expression evaluates to true, in which case the
       binding is created and successful.

   3. The value of the boolean expression is undefined. In this case,
       the binding is also created and successful.

   In all three cases, any undefined term becomes bound by the
   expression. Generally speaking, an "ENFORCE" statement is either
   being used as an assignment (condition 3 above) or being used to test
   if a particular format is usable, as is the case with conditions 1
   and 2.

4.10. Formal Specification of Field Lengths

   In many of the examples each field has been followed by a comment
   indicating the length of the field. Indicating the length of a field
   like this is optional, but can be very helpful for the reader.
   However, whilst useful to the reader, comments have no formal
   meaning.

   One of the most common uses for "ENFORCE" statements (see
   Section 4.9) is to explicitly define the length of a field within a
   header. Using "ENFORCE" statements for this purpose has formal
   meaning but is not so easy to read. Therefore, an abbreviated form
   is provided for this use of "ENFORCE", which is both easy to read and
   has formal meaning.

   An expression defining the length of a field can be specified in
   square brackets after the appearance of that field in a format. If
   the field can take several alternative lengths, then the expressions
   defining those lengths can be enumerated as a comma separated list
   within the square brackets. For example:

     field_1   [4];
     field_2   [a+b, 2];
     field_3 =:= lsb(16, 16) [26];

   The actual length attribute, which is bound by this notation, depends
   on whether it appears in a "COMPRESSED", "UNCOMPRESSED", or "CONTROL"
   field list (see Section 4.12.1 and its subsections). In a
   "COMPRESSED" field list, the field's "CLENGTH" attribute is bound.
   In "UNCOMPRESSED" and "CONTROL" field lists, the field's "ULENGTH"
   attribute is bound. Abbreviated "ENFORCE" statements are not allowed
   in "DEFAULT" sections (see Section 4.12.1.5). Therefore, the above
   notation would not be allowed to appear in a "DEFAULT" section.
   However, if the above appeared in an "UNCOMPRESSED" or "CONTROL"
   section, it would be equivalent to:

```
   field_1;                   ENFORCE(field_1.ULENGTH == 4);
   field_2;                   ENFORCE((field_2.ULENGTH == 2)
                                   || (field_2.ULENGTH == a+b));
   field_3 =:= lsb(16, 16); ENFORCE(field_3.ULENGTH == 26);
```

A special case exists for fields that have a variable length that the
notator does not wish, or is not able to, define using an expression.
The keyword "VARIABLE" can be used in the following case:

```
   variable_length_field  [ VARIABLE ];
```

Formally, this provides no restrictions on the field length, but maps
onto any positive integer or to a value of zero. It will therefore
be necessary to define the length of the field elsewhere (see the
final paragraphs of Section 4.12.1.1 and Section 4.12.1.2). This may
either be in the notation or in the English text of the profile
within which the FN is contained. Within the square brackets, the
keyword "VARIABLE" may be used as a term in an expression, just like
any other term that normally appears in an expression. For example:

```
      field  [ 8 * (5 + VARIABLE) ];
```

This defines a field whose length is a whole number of octets and at
least 40 bits (5 octets).

4.11. Library of Encoding Methods

A number of common techniques for compressing header fields are
defined as part of the ROHC-FN library so that they can be reused
when creating new ROHC-FN specifications. Their notation is
described below.

As an alternative, or a complement, to this library of encoding
methods, a ROHC-FN specification can define its own set of encoding
methods, using the formal notation (see Section 4.12) or using a
textual definition (see Section 4.13).

4.11.1. uncompressed_value

The "uncompressed_value" encoding method is used to encode header
fields for which the uncompressed value can be defined using a
mathematical expression (including constant values). This encoding
method is defined as follows:

```
   uncompressed_value(len, val) {
     UNCOMPRESSED {
       field;
       ENFORCE(field.ULENGTH == len);
       ENFORCE(field.UVALUE == val);
     }
     COMPRESSED {
       field;
       ENFORCE(field.CLENGTH == 0);
     }
   }
```

To exemplify the usage of "uncompressed_value" encoding, the IPv6
header version number is a 4-bit field that always has the value 6:

```
   version   =:=   uncompressed_value(4, 6);
```

Here is another example of value encoding, using an expression to
calculate the length:

```
   padding =:= uncompressed_value(nbits - 8, 0);
```

The expression above uses an encoding method parameter, "nbits", that
in this example specifies how many significant bits there are in the
data to calculate how many pad bits to use. See Section 4.12.2 for
more information on encoding method parameters.

4.11.2. compressed_value

The "compressed_value" encoding method is used to define fields in
compressed formats for which there is no counterpart in the
uncompressed format (i.e., control fields). It can be used to
specify compressed fields whose value can be defined using a
mathematical expression (including constant values). This encoding
method is defined as follows:

```
   compressed_value(len, val) {
     UNCOMPRESSED {
       field;
       ENFORCE(field.ULENGTH == 0);
     }
     COMPRESSED {
       field;
       ENFORCE(field.CLENGTH == len);
       ENFORCE(field.CVALUE == val);
     }
   }
```

   One possible use of this encoding method is to define padding in a
   compressed format:

     pad_to_octet_boundary   =:=   compressed_value(3, 0);

   A more common use is to define a discriminator field to make it
   possible to differentiate between different compressed formats within
   an encoding method (see Section 4.12). For convenience, the notation
   provides syntax for specifying "compressed_value" encoding in the
   form of a binary string. The binary string to be encoded is simply
   given in single quotes; the "CLENGTH" attribute of the field binds
   with the number of bits in the string, while its "CVALUE" attribute
   binds with the value given by the string. For example:

     discriminator   =:=   '01101';

   This has exactly the same meaning as:

     discriminator   =:=   compressed_value(5, 13);

4.11.3. irregular

   The "irregular" encoding method is used to encode a field in the
   compressed format with a bit pattern identical to the uncompressed
   field. This encoding method is defined as follows:

```
     irregular(len) {
       UNCOMPRESSED {
         field;
         ENFORCE(field.ULENGTH == len);
       }
       COMPRESSED {
         field;
         ENFORCE(field.CLENGTH == len);
         ENFORCE(field.CVALUE == field.UVALUE);
       }
     }
```

   For example, the checksum field of the TCP header is a 16-bit field
   that does not follow any predictable pattern from one header to
   another (and so it cannot be compressed):

     tcp_checksum =:=   irregular(16);

   Note that the length does not have to be constant, for example, an
   expression can be used to derive the length of the field from the
   value of another field.

4.11.4. static

   The "static" encoding method compresses a field whose length and
   value are the same as for a previous header in the flow, i.e., where
   the field completely matches an existing entry in the context:

     field   =:=   static;

   The field's "UVALUE" and "ULENGTH" attributes bind with their
   respective values in the context and the "CLENGTH" attribute is bound
   to zero.

   Since the field value is the same as a previous field value, the
   entire field can be reconstructed from the context, so it is
   compressed to zero bits and does not appear in the compressed format.

   For example, the source port of the TCP header is a field whose value
   does not change from one packet to the next for a given flow:

     src_port =:=   static;

4.11.5. lsb

   The least significant bits encoding method, "lsb", compresses a field
   whose value differs by a small amount from the value stored in the
   context. The least significant bits of the field value are
   transmitted instead of the original field value.

     field =:=   lsb(<num_lsbs_param>, <offset_param>);

   Here, "num_lsbs_param" is the number of least significant bits to
   use, and "offset_param" is the interpretation interval offset as
   defined below.

   The parameter "num_lsbs_param" binds with the "CLENGTH" attribute,
   the "UVALUE" attribute binds to the value within the interval whose
   least significant bits match the "CVALUE" attribute. The value of
   the "ULENGTH" can be derived from the information stored in the
   context.

   For example, the TCP sequence number:

     tcp_sequence_number   =:=   lsb(14, 8192);

   This takes up 14 bits, and can communicate any value that is between
   8192 lower than the value of the field stored in context and 8191
   above it.

The interpretation interval can be described as a function of a value
stored in the context, ref_value, and of num_lsbs_param:

   f(context_value, num_lsbs_param) = [ref_value - offset_param,
             ref_value + (2^num_lsbs_param - 1) - offset_param]

where offset_param is an integer.

```
       <-- interpretation interval (size is 2^num_lsbs_param) -->
        |---------------------------+---------------------------|
      lower                      ref_value                    upper
      bound                                                   bound
```

where:

```
      lower bound = ref_value - offset_param
      upper bound = ref_value + (2^num_lsbs_param-1) - offset_param
```

The "lsb" encoding method can therefore compress a field whose value
lies between the lower and the upper bounds, inclusively, of the
interpretation interval. In particular, if offset_param = 0, then
the field value can only stay the same or increase relative to the
reference value ref_value. If offset_param = -1, then it can only
increase, whereas if offset_param = 2^num_lsbs_param, then it can
only decrease.

The compressed field takes up the specified number of bits in the
compressed format (i.e., num_lsbs_param).

The compressor may not be able to determine the exact reference value
stored in the decompressor context and that will be used by the
decompressor, since some packets that would have updated the context
may have been lost or damaged. However, from feedback received or by
making assumptions, the compressor can limit the candidate set of
values. The compressor can then select a format that uses "lsb"
encoding, defined with suitable values for its parameters
num_lsbs_param and offset_param, such that no matter which context
value in the candidate set the decompressor uses, the resulting
decompression is correct. If that is not possible, the "lsb"
encoding method fails (which typically results in a less efficient
compressed format being chosen by the compressor). How the
compressor determines what reference values it stores and maintains
in its set of candidate references is outside the scope of the
notation.

4.11.6. crc

   The "crc" encoding method provides a CRC calculated over a block of
   data. The algorithm used to calculate the CRC is the one specified
   in [RFC4995]. The "crc" method takes a number of parameters:

   o the number of bits for the CRC (crc_bits),

   o the bit-pattern for the polynomial (bit_pattern),

   o the initial value for the CRC register (initial_value),

   o the value of the block of data, represented using either the
      "UVALUE" or "CVALUE" attribute of a field (block_data_value); and

   o the size in octets of the block of data (block_data_length).

   That is:

      field   =:=   crc(<num_bits>, <bit_pattern>, <initial_value>,
                        <block_data_value>, <block_data_length>);

   When specifying the bit pattern for the polynomial, each bit
   represents the coefficient for the corresponding term in the
   polynomial. Note that the highest order term is always present (by
   definition) and therefore does not need specifying in the bit
   pattern. Therefore, a CRC polynomial with n terms in it is
   represented by a bit pattern with n-1 bits set.

   The CRC is calculated in least significant bit (LSB) order.

   For example:

      // 3 bit CRC, C(x) = x^0 + x^1 + x^3
      crc_field =:= crc(3, 0x6, 0xF, THIS.CVALUE, THIS.CLENGTH);

   Usage of the "THIS" keyword (see Section 4.6) as shown above, is
   typical when using "crc" encoding. For example, when used in the
   encoding method for an entire header, it causes the CRC to be
   calculated over all fields in the header.

4.12. Definition of Encoding Methods

   New encoding methods can be defined in a formal specification. These
   compose groups of individual fields into a contiguous block.

   Encoding methods have names and may have parameters; they can also be
   used in the same way as any other encoding method from the library of

encoding methods. Since they can contain references to other
encoding methods, complicated formats can be broken down into
manageable pieces in a hierarchical fashion.

This section describes the various features used to define new
encoding methods.

4.12.1. Structure

This simplest form of defining an encoding method is to specify a
single encoding. For example:

```
compound_encoding_method
{
  UNCOMPRESSED {
    field_1;  //  4 bits
    field_2;  // 12 bits
  }

  COMPRESSED {
    field_2 =:= uncompressed_value(12, 9); //  0 bits
    field_1 =:= irregular(4);              //  4 bits
  }
}
```

The above begins with the new method's identifier,
"compound_encoding_method". The definition of the method then
follows inside curly brackets, "{" and "}". The first item in the
definition is the "UNCOMPRESSED" field list, which gives the order of
the fields in the uncompressed format. This is followed by the
compressed format field list ("COMPRESSED"). This list gives the
order of fields in the compressed format and also gives the encoding
method for each field.

In the example, both the formats list each field exactly once.
However, sometimes it is necessary to specify more than one binding
for a given field, which means it appears more than once in the field
list. In this case, it is the first occurrence of the field in the
list that indicates its position in the field order. The subsequent
occurrences of the field only specify binding information, not field
order information.

The different components of this example are described in more detail
below. Other components that can be used in the definition of
encoding methods are also defined thereafter.

4.12.1.1. Uncompressed Format - "UNCOMPRESSED"

   The uncompressed field list is defined by "UNCOMPRESSED", which
   specifies the fields of the uncompressed format in the order that
   they appear in the uncompressed header. The sum of the lengths of
   each individual uncompressed field in the list must be equal to the
   length of the field being encoded. Finally, the representation of
   the uncompressed format described using the list of fields in the
   "UNCOMPRESSED" section, for which compressed formats are being
   defined, always consists of one single contiguous block of bits.

   In the example above in Section 4.12.1, the uncompressed field list
   is "field_1", followed by "field_2". This means that a field being
   encoded by this method is divided into two subfields, "field_1" and
   "field_2". The total uncompressed length of these two fields
   therefore equals the length of the field being encoded:

     field_1.ULENGTH + field_2.ULENGTH == THIS.ULENGTH

   In the example, there are only two fields, but any number of fields
   may be used. This relationship applies to however many fields are
   actually used. Any arrangement of fields that efficiently describes
   the content of the uncompressed header may be chosen -- this need not
   be the same as the one described in the specifications for the
   protocol header being compressed.

   For example, there may be a protocol whose header contains a 16-bit
   sequence number, but whose sessions tend to be short-lived. This
   would mean that the high bits of the sequence number are almost
   always constant. The "UNCOMPRESSED" format could reflect this by
   splitting the original uncompressed field into two fields, one field
   to represent the almost-always-zero part of the sequence number, and
   a second field to represent the salient part.

   An "UNCOMPRESSED" field list may specify encoding methods in the same
   way as the "COMPRESSED" field list in the example. Encoding methods
   specified therein are used whenever a packet with that uncompressed
   format is being encoded. The encoding of a packet with a given
   uncompressed format can only succeed if all of its encoding methods
   and "ENFORCE" statements succeed (see Section 4.9).

   The total length of each uncompressed format must always be defined.
   The length of each of the fields in an uncompressed format must also
   be defined. This means that the bindings in the "UNCOMPRESSED",
   "COMPRESSED" (see Section 4.12.1.2 below), "CONTROL" (see
   Section 4.12.1.3 below), "INITIAL" (see Section 4.12.1.4 below), and
   "DEFAULT" (see Section 4.12.1.5 below) field lists must, between
   them, define the "ULENGTH" attribute of every field in an

uncompressed format so that there is an unambiguous mapping from the
bits in the uncompressed format to the fields listed in the
"UNCOMPRESSED" field list.

4.12.1.2. Compressed Format - "COMPRESSED"

Similar to the uncompressed field list, the fields in the compressed
header will appear in the order specified by the compressed field
list given for a compressed format. Each individual field is encoded
in the manner given for that field. The total length of the
compressed data will be the sum of the compressed lengths of all the
individual fields. In the example from Section 4.12.1, the encoding
methods used for these fields indicate that they are zero and 4 bits
long, making a total of 4 bits.

The order of the fields specified in a "COMPRESSED" field list does
not have to match the order they appear in the "UNCOMPRESSED" field
list. It may be desirable to reorder the fields in the compressed
format to align the compressed header to the octet boundary, or for
other reasons. In the above example, the order is in fact the
opposite of that in the uncompressed format.

The compressed field list specifies that the encoding for "field_1"
is "irregular", and takes up 4 bits in both the compressed format and
uncompressed format. The encoding for "field_2" is
"uncompressed_value", which means that the field has a fixed value,
so it can be compressed to zero bits. The value it takes is 9, and
it is 12 bits wide in the uncompressed format.

Fields like "field_2", which compress to zero bits in length, may
appear anywhere in the field list without changing the compressed
format because their position in the list is not significant. In
fact, if the encoding method for this field were defined elsewhere
(for example, in the "UNCOMPRESSED" section), this field could be
omitted from the "COMPRESSED" section altogether:

```
compound_encoding_method
{
  UNCOMPRESSED {
    field_1;                              //  4 bits
    field_2 =:= uncompressed_value(12, 9);  // 12 bits
  }

  COMPRESSED {
    field_1 =:= irregular(4);             //  4 bits
  }
}
```

   The total length of each compressed format must always be defined.
   The length of each of the fields in a compressed format must also be
   defined. This means that the bindings in the "UNCOMPRESSED",
   "COMPRESSED", "CONTROL" (see Section 4.12.1.3 below), "INITIAL" (see
   Section 4.12.1.4 below), and "DEFAULT" (see Section 4.12.1.5 below)
   field lists must between them define the "CLENGTH" attribute of every
   field in a compressed format so that there is an unambiguous mapping
   from the bits in the compressed format to the fields listed in the
   "COMPRESSED" field list.

4.12.1.3. Control Fields - "CONTROL"

   Control fields are defined using the "CONTROL" field list. The
   control field list specifies all fields that do not appear in the
   uncompressed format, but that have an uncompressed value
   (specifically those with an "ULENGTH" greater than zero). Such
   fields may be used to help compress fields from the uncompressed
   format more efficiently. A control field could be used to improve
   efficiency by representing some commonality between a number of the
   uncompressed fields, or by representing some information about the
   flow that is not explicitly contained in the protocol headers.

   For example in IPv4, the behaviour of the IP-ID field in a flow
   varies depending on how the endpoints handle IP-IDs. Sometimes the
   behaviour is effectively random and sometimes the IP-ID follows a
   predictable sequence. The type of IP-ID behaviour is information
   that is never communicated explicitly in the uncompressed header.

   However, a profile can still be designed to identify the behaviour
   and adjust the compression strategy according to the identified
   behaviour, thereby improving the compression performance. To do so,
   the ROHC-FN specification can introduce an explicit field to
   communicate the IP-ID behaviour in compressed format -- this is done
   by introducing a control field:

```
     ipv4
     {
       UNCOMPRESSED {
         version;        // 4 bits
         hdr_length;     // 4 bits
         protocol;       // 8 bits
         dscp;           // 6 bits
         ip_ecn_flags;   // 2 bits
         ttl_hopl;       // 8 bits
         df;             // 1 bit
         mf;             // 1 bit
         rf;             // 1 bit
         frag_offset;    // 13 bits
```

```
      ip_id;          // 16 bits
      src_addr;       // 32 bits
      dst_addr;       // 32 bits
      checksum;       // 16 bits
      length;         // 16 bits
    }

    CONTROL {
      ip_id_behavior; // 1 bit
          :
          :
```

The "CONTROL" field list is equivalent to the "UNCOMPRESSED" field
list for fields that do not appear in the uncompressed format. It
defines a field that has the same properties (the same defined
attributes, etc.) as fields appearing in the uncompressed format.

Control fields are initialised by using the appropriate encoding
methods and/or by using "ENFORCE" statements. This may be done
inside the "CONTROL" field list.

For example:

```
  example_encoding_method_definition
  {
    UNCOMPRESSED {
      field_1 =:= some_encoding;
    }

    CONTROL {
      scaled_field;
      ENFORCE(scaled_field.UVALUE == field_1.UVALUE / 8);
      ENFORCE(scaled_field.ULENGTH == field_1.ULENGTH - 3);
    }

    COMPRESSED {
      scaled_field =:= lsb(4, 0);
    }
  }
```

This control field is used to scale down a field in the uncompressed
format by a factor of 8 before encoding it with the "lsb" encoding
method. Scaling it down makes the "lsb" encoding more efficient.

Control fields may also be used with a global scope. In this case,
their declaration must be outside of any encoding method definition.
They are then visible within any encoding method, thus allowing
information to be shared between encoding methods directly.

4.12.1.4. Initial Values - "INITIAL"

   In order to allow fields in the very first usage of a specific format
   to be compressed with "static", "lsb", or other encoding methods that
   depend on the context, it is possible to specify initial bindings for
   such fields. This is done using "INITIAL", for example:

      INITIAL {
         field =:= uncompressed_value(4, 6);
      }

   This initialises the "UVALUE" of "field" to 6 and initialises its
   "ULENGTH" to 4. Unlike all other bindings specified in the formal
   notation, these bindings are applied to the context of the field, if
   the field's context is undefined. This is particularly useful when
   using encoding methods that rely on context being present, such as
   "static" or "lsb", with the first packet in a flow.

   Because the "INITIAL" field list is used to bind the context alone,
   it makes no sense to specify initial bindings that themselves rely on
   the context, for example, "lsb". Such usage is not allowed.

4.12.1.5. Default Field Bindings - "DEFAULT"

   Default bindings may be specified for each field or attribute. The
   default encoding methods specify the encoding method to use for a
   field if no binding is given elsewhere for the value of that field.
   This is helpful to keep the definition of the formats concise, as the
   same encoding method need not be repeated for every format, when
   defining multiple formats (see Section 4.12.3).

   Default bindings are optional and may be given for any combination of
   fields and attributes which are in scope.

   The syntax for specifying default bindings is similar to that used to
   specify a compressed or uncompressed format. However, the order of
   the fields in the field list does not affect the order of the fields
   in either the compressed or uncompressed format. This is because the
   field order is specified individually for each "COMPRESSED" format
   and "UNCOMPRESSED" format.

   Here is an example:

      DEFAULT {
         field_1 =:= uncompressed_value(4, 1);
         field_2 =:= uncompressed_value(4, 2);
         field_3 =:= lsb(3, -1);
         ENFORCE(field_4.ULENGTH == 4);

```
   }
```

   Here default bindings are specified for fields 1 to 3. A default
   binding for the "ULENGTH" attribute of field_4 is also specified.

   Fields for which there is a default encoding method do not need their
   bindings to be specified in the field list of any format that uses
   the default encoding method for that field. Any format that does not
   use the default encoding method must explicitly specify a binding for
   the value of that field's attributes.

   If elsewhere a binding is not specified for the attributes of a
   field, the default encoding method is used. If the default encoding
   method always compresses the field down to zero bits, the field can
   be omitted from the compressed format's field list. Like any other
   zero-bit field, its position in the field list is not significant.

   The "DEFAULT" field list may contain default bindings for individual
   attributes by using "ENFORCE" statements. A default binding for an
   individual attribute will only be used if elsewhere there is no
   binding given for that attribute or the field to which it belongs.
   If elsewhere there is an "ENFORCE" statement binding that attribute,
   or an encoding method binding the field to which it belongs, the
   default binding for the attribute will not be used. This applies
   even if the specified encoding method does not bind the particular
   attribute given in the "DEFAULT" section. However, an "ENFORCE"
   statement elsewhere that only binds the length of the field still
   allows the default bindings to be used, except for default "ENFORCE"
   statements which bind nothing but the field's length.

   To clarify, assuming the default bindings given in the example above,
   the first three of the following four compressed formats would not
   use the default binding for "field_4.ULENGTH":

```
      COMPRESSED format1 {
        ENFORCE(field_4.ULENGTH == 3); // set ULENGTH to 3
        ENFORCE(field_4.UVALUE == 7);  // set UVALUE to 7
      }

      COMPRESSED format2 {
        field_4 =:= irregular(3);      // set ULENGTH to 3
      }

      COMPRESSED format3 {
        field_4 =:= '1010';            // set ULENGTH to zero
      }
```

```
COMPRESSED format4 {
  ENFORCE(field_4.UVALUE == 12); // use default ULENGTH
}
```

The fourth format is the only one that uses the default binding for
"field_4.ULENGTH".

In summary, the default bindings of an encoding method are only used
for formats that do not already specify a binding for the value of
all of their fields. For the formats that do use default bindings,
only those fields and attributes whose bindings are not specified are
looked up in the "DEFAULT" field list.

4.12.2. Arguments

Encoding methods may take arguments that control the mapping between
compressed and uncompressed fields. These are specified immediately
after the method's name, in parentheses, as a comma-separated list.

For example:

```
poor_mans_lsb(variable_length)
{
  UNCOMPRESSED {
    constant_bits;
    variable_bits;
  }

  COMPRESSED {
    variable_bits =:= irregular(variable_length);
    constant_bits =:= static;
  }
}
```

As with any encoding method, all arguments take individual values,
such as an integer literal or a field attribute, rather than entire
fields. Although entire fields cannot be passed as arguments, it is
possible to pass each of their attributes instead, which is
equivalent.

Recall that all bindings are two-way, so that rather than the
arguments acting as "inputs" to the encoding method, the result of an
encoding method may be to bind the parameters passed to it.

For example:

```
set_to_double(arg1, arg2)
{
  CONTROL {
    ENFORCE(arg1 == 2 * arg2);
  }
}
```

This encoding method will attempt to bind the first argument to twice
the value of the second. In fact this "encoding" method is
pathological. Since it defines no fields, it does not do any actual
encoding at all. "CONTROL" sections are more appropriate to use for
this purpose than "UNCOMPRESSED".

4.12.3. Multiple Formats

Encoding methods can also define multiple formats for a given header.
This allows different compression methods to be used depending on
what is the most efficient way of compressing a particular header.

For example, a field may have a fixed value most of the time, but the
value may occasionally change. Using a single format for the
encoding, this field would have to be encoded using "irregular" (see
Section 4.11.3), even though the value only changes rarely. However,
by defining multiple formats, we can provide two alternative
encodings: one for when the value remains fixed and another for when
the value changes.

This is the topic of the following sub-sections.

4.12.3.1. Naming Convention

When compressed formats are defined, they must be defined using the
reserved word "COMPRESSED". Similarly, uncompressed formats must be
defined using the reserved word "UNCOMPRESSED". After each of these
keywords, a name may be given for the format. If no name is given to
the format, the name of the format is empty.

Format names, except for the case where the name is empty, follow the
syntactic rules of identifiers as described in Section 4.2.

Format names must be unique within the scope of the encoding method
to which they belong, except for the empty name, which may be used
for one "COMPRESSED" and one "UNCOMPRESSED" format.

4.12.3.2. Format Discrimination

   Each of the compressed formats has its own field list. A compressor
   may pick any of these alternative formats to compress a header, as
   long as the field bindings it employs can be used with the
   uncompressed format. For example, the compressor could not choose to
   use a compressed format that had a "static" encoding for a field
   whose "UVALUE" attribute differs from its corresponding value in the
   context.

   More formally, the compressor can choose any combination of an
   uncompressed format and a compressed format for which no binding for
   any of the field's attributes "fail", i.e., the encoding methods and
   "ENFORCE" statements (see Section 4.9) that bind their compressed
   attributes succeed. If there are multiple successful combinations,
   the compressor can choose any one. Otherwise if there are no
   successful combinations, the encoding method "fails". A format will
   never fail due to it not defining the "UVALUE" attribute of a field.
   A format only fails if it fails to define one of the compressed
   attributes of one of the fields in the compressed format, or leaves
   the length of the uncompressed format undefined.

   Because the compressor has a choice, it must be possible for the
   decompressor to discriminate between the different compressed formats
   that the compressor could have chosen. A simple approach to this
   problem is for each compressed format to include a "discriminator"
   that uniquely identifies that particular "COMPRESSED" format. A
   discriminator is a control field; it is not derived from any of the
   uncompressed field values (see Section 4.11.2).

4.12.3.3. Example of Multiple Formats

   Putting this all together, here is a complete example of the
   definition of an encoding method with multiple compressed formats:

```
     example_multiple_formats
     {
       UNCOMPRESSED {
         field_1;  //  4 bits
         field_2;  //  4 bits
         field_3;  // 24 bits
       }

       DEFAULT {
         field_1 =:= static;
         field_2 =:= uncompressed_value(4, 2);
         field_3 =:= lsb(4, 0);
       }
```

```
      COMPRESSED format0 {
        discriminator =:= '0'; // 1 bit
        field_3;                // 4 bits
      }

      COMPRESSED format1 {
        discriminator =:= '1';            //  1 bit
        field_1      =:= irregular(4);  //  4 bits
        field_3      =:= irregular(24); // 24 bits
      }
    }
```

   Note the following:

   o "field_1" and "field_3" both have default encoding methods
      specified for them, which are used in "format0", but are
      overridden in "format1"; the default encoding method of "field_2"
      however, is not overridden.

   o "field_1" and "field_2" have default encoding methods that
      compress to zero bits. When these are used in "format0", the
      field names do not appear in the field list.

   o "field_3" has an encoding method that does not compress to zero
      bits, so whilst "field_3" has no encoding specified for it in the
      field list of "format0", it still needs to appear in the field
      list to specify where it goes in the compressed format.

   o In the example, all the fields in the uncompressed format have
      default encoding methods specified for them, but this is not a
      requirement. Default encodings can be specified for only some or
      even none of the fields of the uncompressed format.

   o In the example, all the default encoding methods are on fields
      from the uncompressed format, but this is not a requirement.
      Default encoding methods can be specified for control fields.

4.13. Profile-Specific Encoding Methods

   The library of encoding methods defined by ROHC-FN in Section 4.11
   provides a basic and generic set of field encoding methods. When
   using a ROHC-FN specification in a ROHC profile, some additional
   encodings specific to the particular protocol header being compressed
   may, however, be needed, such as methods that infer the value of a
   field from other values.

   These methods are specific to the properties of the protocol being
   compressed and will thus have to be defined within the profile

specification itself. Such profile-specific encoding methods,
defined either in ROHC-FN syntax or rigorously in plain text, can be
referred to in the ROHC-FN specification of the profile's formats in
the same way as any method in the ROHC-FN library.

Encoding methods that are not defined in the formal notation are
specified by giving their name, followed by a short description of
where they are defined, in double quotes, and a semi-colon.

For example:

    inferred_ip_v4_header_checksum "defined in RFCxxxx Section 6.4.1";

5. Security Considerations

This document describes a formal notation similar to ABNF [RFC4234],
and hence is not believed to raise any security issues (note that
ABNF has a completely separate purpose to the ROHC formal notation).

6. Contributors

Richard Price did much of the foundational work on the formal
notation. He authored the initial document describing a formal
notation on which this document is based.

Kristofer Sandlund contributed to this work by applying new ideas to
the ROHC-TCP profile, by providing feedback, and by helping resolve
different issues during the entire development of the notation.

Carsten Bormann provided the translation of the formal notation
syntax using ABNF in Appendix A, and also contributed with feedback
and reviews to validate the completeness and correctness of the
notation.

7. Acknowledgements

A number of important concepts and ideas have been borrowed from ROHC
[RFC3095].

Thanks to Mark West, Eilert Brinkmann, Alan Ford, and Lars-Erik
Jonsson for their contributions, reviews, and feedback that led to
significant improvements to the readability, completeness, and
overall quality of the notation.

Thanks to Stewart Sadler, Caroline Daniels, Alan Finney, and David
Findlay for their reviews and comments. Thanks to Rob Hancock and
Stephen McCann for their early work on the formal notation. The

authors would also like to thank Christian Schmidt, Qian Zhang,
Hongbin Liao, and Max Riegel for their comments and valuable input.

Additional thanks: this document was reviewed during working group
last-call by committed reviewers Mark West, Carsten Bormann, and Joe
Touch, as well as by Sally Floyd who provided a review at the request
of the Transport Area Directors. Thanks also to Magnus Westerlund
for his feedback in preparation for the IESG review.

8. References

8.1. Normative References

   [C90]   ISO/IEC, "ISO/IEC 9899:1990 Information technology --
               Programming Language C", ISO 9899:1990, April 1990.

   [RFC2822]   Resnick, P., Ed., "STANDARD FOR THE FORMAT OF ARPA
               INTERNET TEXT MESSAGES", RFC 2822, April 2001.

   [RFC4234]   Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax
               Specifications: ABNF", RFC 4234, October 2005.

   [RFC4995]   Jonsson, L-E., Pelletier, G., and K. Sandlund, "The RObust
               Header Compression (ROHC) Framework", RFC 4995, July 2007.

8.2. Informative References

   [RFC3095]   Bormann, C., Burmeister, C., Degermark, M., Fukushima, H.,
               Hannu, H., Jonsson, L-E., Hakenberg, R., Koren, T., Le,
               K., Liu, Z., Martensson, A., Miyazaki, A., Svanbro, K.,
               Wiebke, T., Yoshimura, T., and H. Zheng, "RObust Header
               Compression (ROHC): Framework and four profiles: RTP, UDP,
               ESP, and uncompressed", RFC 3095, July 2001.

   [RFC791]   University of Southern California, "DARPA INTERNET PROGRAM
               PROTOCOL SPECIFICATION", RFC 791, September 1981.

Appendix A. Formal Syntax of ROHC-FN

   This section gives a definition of the syntax of ROHC-FN in ABNF
   [RFC4234], using "fnspec" as the start rule.

```
; overall structure
fnspec    = S *(constdef S) [globctl S] 1*(methdef S)
constdef  = constname S "=" S expn S ";"
globctl   = CONTROL S formbody
methdef   = id S [parmlist S] "{" S 1*(formatdef S) "}"
          / id S [parmlist S] STRQ *STRCHAR STRQ S ";"
parmlist  = "(" S id S *( "," S id S ) ")"
formatdef = formhead S formbody
formhead  = UNCOMPRESSED [ 1*WS id ]
          / COMPRESSED [ 1*WS id ]
          / CONTROL / INITIAL / DEFAULT
formbody  = "{" S *((fielddef/enforcer) S) "}"
fielddef  = fieldgroup S ["=:=" S encspec S] [lenspec S] ";"
fieldgroup = fieldname *( S ":" S fieldname )
fieldname = id
encspec   = "'" *("0"/"1") "'"
          / id [ S "(" S expn S *( "," S expn S ) ")"]
lenspec   = "[" S expn S *("," S expn S) "]"
enforcer  = ENFORCE S "(" S expn S ")" S ";"


; expressions
expn  = *(expnb S "||" S) expnb
expnb = *(expna S "&&" S) expna
expna = *(expn7 S ("=="/"!=") S) expn7
expn7 = *(expn6 S ("<"/"<="/">"/">=") S) expn6
expn6 = *(expn4 S ("+"/"-") S) expn4
expn4 = *(expn3 S ("*"/"/"/"%") S) expn3
expn3 = expn2 [S "^" S expn3]
expn2 = ["!" S] expn1
expn1 = expn0 / attref / constname / litval / id
expn0 = "(" S expn S ")" / VARIABLE
attref       = fieldnameref "." attname
fieldnameref = fieldname / THIS
attname      = ( U / C ) ( LENGTH / VALUE )
litval       = ["-"] "0b" 1*("0"/"1")
             / ["-"] "0x" 1*(DIGIT/"a"/"b"/"c"/"d"/"e"/"f")
             / ["-"] 1*DIGIT
             / false / true
```

```
   ; lexical categories
   constname = UPCASE *(UPCASE / DIGIT / "_")
   id        = ALPHA *(ALPHA / DIGIT / "_")
   ALPHA     = %x41-5A / %x61-7A
   UPCASE    = %x41-5A
   DIGIT     = %x30-39
   COMMENT   = "//" *(SP / HTAB / VCHAR) CRLF
   SP        = %x20
   HTAB      = %x09
   VCHAR     = %x21-7E
   CRLF      = %x0A / %x0D.0A
   NL        = COMMENT / CRLF
   WS        = SP / HTAB / NL
   S         = *WS
   STRCHAR   = SP / HTAB / %x21 / %x23-7E
   STRQ      = %x22


   ; case-sensitive literals
   C            = %d67
   COMPRESSED   = %d67.79.77.80.82.69.83.83.69.68
   CONTROL      = %d67.79.78.84.82.79.76
   DEFAULT      = %d68.69.70.65.85.76.84
   ENFORCE      = %d69.78.70.79.82.67.69
   INITIAL      = %d73.78.73.84.73.65.76
   LENGTH       = %d76.69.78.71.84.72
   THIS         = %d84.72.73.83
   U            = %d85
   UNCOMPRESSED = %d85.78.67.79.77.80.82.69.83.83.69.68
   VALUE        = %d86.65.76.85.69
   VARIABLE     = %d86.65.82.73.65.66.76.69
   false        = %d102.97.108.115.101
   true         = %d116.114.117.101
```

Appendix B. Bit-level Worked Example

   This section gives a worked example at the bit level, showing how a
   simple ROHC-FN specification describes the compression of real data
   from an imaginary protocol header. The example used has been kept
   fairly simple, whilst still aiming to illustrate some of the
   intricacies that arise in use of the notation. In particular, fields
   have been kept short to make it possible to read the binary
   representation of the headers without too much difficulty.

B.1. Example Packet Format

   Our imaginary header is just 16 bits long, and consists of the
   following fields:

   1. version number -- 2 bits

   2. type -- 2 bits

   3. flow id -- 4 bits

   4. sequence number -- 4 bits

   5. flag bits -- 4 bits

   So for example 0101000100010000 indicates a header with a version
   number of one, a type of one, a flow id of one, a sequence number of
   one, and all flag bits set to zero.

   Here is an ASCII box notation diagram of the imaginary header:

```
   0   1   2   3   4   5   6   7
 +---+---+---+---+---+---+---+---+
 |version| type  |    flow_id    |
 +---+---+---+---+---+---+---+---+
 |   sequence_no |   flag_bits   |
 +---+---+---+---+---+---+---+---+
```

B.2. Initial Encoding

   An initial definition based solely on the above information is as
   follows:

```
     eg_header
     {
       UNCOMPRESSED {
         version_no  [ 2 ];
         type        [ 2 ];
         flow_id     [ 4 ];
         sequence_no [ 4 ];
         flag_bits   [ 4 ];
       }

       COMPRESSED initial_definition {
         version_no  =:= irregular(2);
         type        =:= irregular(2);
         flow_id     =:= irregular(4);
         sequence_no =:= irregular(4);
         flag_bits   =:= irregular(4);
       }
     }
```

   This defines the format nicely, but doesn't actually offer any
   compression. If we use it to encode the above header, we get:

```
     Uncompressed header: 0101000100010000
     Compressed header:   0101000100010000
```

   This is because we have stated that all fields are "irregular" --
   i.e., we haven't specified anything about their behaviour.

   Note that since we have only one compressed format and one
   uncompressed format, it makes no difference whether the encoding
   methods for each field are specified in the compressed or
   uncompressed format. It would make no difference at all if we wrote
   the following instead:

```
     eg_header
     {
       UNCOMPRESSED {
         version_no  =:= irregular(2);
         type        =:= irregular(2);
         flow_id     =:= irregular(4);
         sequence_no =:= irregular(4);
         flag_bits   =:= irregular(4);
       }
```

```
        COMPRESSED initial_definition {
          version_no   [ 2 ];
          type         [ 2 ];
          flow_id      [ 4 ];
          sequence_no  [ 4 ];
          flag_bits    [ 4 ];
        }
      }
```

B.3. Basic Compression

   In order to achieve any compression we need to notate more knowledge
   about the header and its behaviour in a flow. For example, we may
   know the following facts about the header:

   1. version number -- indicates which version of the protocol this
       is: always one for this version of the protocol.

   2. type -- may take any value.

   3. flow id -- may take any value.

   4. sequence number -- make take any value.

   5. flag bits -- contains three flags, a, b, and c, each of which may
       be set or clear, and a reserved flag bit, which is always clear
       (i.e., zero).

   We could notate this knowledge as follows:

```
     eg_header
     {
       UNCOMPRESSED {
         version_no     [ 2 ];
         type           [ 2 ];
         flow_id        [ 4 ];
         sequence_no    [ 4 ];
         abc_flag_bits  [ 3 ];
         reserved_flag  [ 1 ];
       }

       COMPRESSED basic {
         version_no    =:= uncompressed_value(2, 1)  [ 0 ];
         type          =:= irregular(2)              [ 2 ];
         flow_id       =:= irregular(4)              [ 4 ];
         sequence_no   =:= irregular(4)              [ 4 ];
         abc_flag_bits =:= irregular(3)              [ 3 ];
         reserved_flag =:= uncompressed_value(1, 0)  [ 0 ];
```

```
      }
   }
```

Using this simple scheme, we have successfully encoded the fact that
one of the fields has a permanently fixed value of one, and therefore
contains no useful information. We have also encoded the fact that
the final flag bit is always zero, which again contains no useful
information. Both of these facts have been notated using the
"uncompressed_value" encoding method (see Section 4.11.1).

Using this new encoding on the above header, we get:

```
   Uncompressed header: 0101000100010000
   Compressed header:   0100010001000
```

This reduces the amount of data we need to transmit by roughly 20%.
However, this encoding fails to take advantage of relationships
between values of a field in one packet and its value in subsequent
packets. For example, every header in the following sequence is
compressed by the same amount despite the similarities between them:

```
   Uncompressed header: 0101000100010000
   Compressed header:   0100010001000


   Uncompressed header: 0101000101000000
   Compressed header:   0100010100000


   Uncompressed header: 0110000101110000
   Compressed header:   1000010111000
```

B.4. Inter-Packet Compression

The profile we have defined so far has not compressed the sequence
number or flow ID fields at all, since they can take any value.
However the value of each of these fields in one header has a very
simple relationship to their values in previous headers:

o the sequence number is unusual -- it increases by three each time,

o the flow_id stays the same -- it always has the same value that it
   did in the previous header in the flow,

o the abc_flag_bits stay the same most of the time -- they usually
   have the same value that they did in the previous header in the
   flow.

An obvious way of notating this is as follows:

```
// This obvious encoding will not work (correct encoding below)
eg_header
{
  UNCOMPRESSED {
    version_no     [ 2 ];
    type           [ 2 ];
    flow_id        [ 4 ];
    sequence_no    [ 4 ];
    abc_flag_bits  [ 3 ];
    reserved_flag  [ 1 ];
  }

  COMPRESSED obvious {
    version_no    =:= uncompressed_value(2, 1);
    type          =:= irregular(2);
    flow_id       =:= static;
    sequence_no   =:= lsb(0, -3);
    abc_flag_bits =:= irregular(3);
    reserved_flag =:= uncompressed_value(1, 0);
  }
}
```

The dependency on previous packets is notated using the "static" and
"lsb" encoding methods (see Section 4.11.4 and Section 4.11.5
respectively). However there are a few problems with the above
notation.

Firstly, and most importantly, the "flow_id" field is notated as
"static", which means that it doesn't change from packet to packet.
However, the notation does not indicate how to communicate the value
of the field initially. There is no point saying "it's the same
value as last time" if there has not been a first time where we
define what that value is, so that it can be referred back to. The
above notation provides no way of communicating that. Similarly with
the sequence number -- there needs to be a way of communicating its
initial value. In fact, except for the explicit notation indicating
their lengths, even the lengths of these two fields would be left
undefined. This problem will be solved below, in Appendix B.5.

Secondly, the sequence number field is communicated very efficiently
in zero bits, but it is not at all robust against packet loss. If a
packet is lost then there is no way to handle the missing sequence
number. When communicating sequence numbers, or any other field
encoded with "lsb" encoding, a very important consideration for the
notator is how robust against packet loss the compressed protocol
should be. This will vary a lot from protocol stack to protocol

stack. For the example protocol we'll assume short, low overhead
flows and say we need to be robust to the loss of just one packet,
which we can achieve with two bits of "lsb" encoding (one bit isn't
enough since the sequence number increases by three each time -- see
Section 4.11.5). This will be addressed below in Appendix B.5.

Finally, although the flag bits are usually the same as in the
previous header in the flow, the profile doesn't make any use of this
fact; since they are sometimes not the same as those in the previous
header, it is not safe to say that they are always the same, so
"static" encoding can't be used exclusively. This problem will be
solved later through the use of multiple formats in Appendix B.6.

B.5. Specifying Initial Values

To communicate initial values for fields compressed with a context
dependent encoding such as "static" or "lsb" we use an "INITIAL"
field list. This can help with fields whose start value is fixed and
known. For example, if we knew that at the start of the flow that
"flow_id" would always be 1 and "sequence_no" would always be 0, we
could notate that like this:

```
  // This encoding will not work either (correct encoding below)
  eg_header
  {
    UNCOMPRESSED {
      version_no     [ 2 ];
      type           [ 2 ];
      flow_id        [ 4 ];
      sequence_no    [ 4 ];
      abc_flag_bits  [ 3 ];
      reserved_flag  [ 1 ];
    }

    INITIAL {
      // set initial values of fields before flow starts
      flow_id     =:= uncompressed_value(4, 1);
      sequence_no =:= uncompressed_value(4, 0);
    }

    COMPRESSED obvious {
      version_no    =:= uncompressed_value(2, 1);
      type          =:= irregular(2);
      flow_id       =:= static;
      sequence_no   =:= lsb(2, -3);
      abc_flag_bits =:= irregular(3);
      reserved_flag =:= uncompressed_value(1, 0);
    }
```

```
   }

   However, this use of "INITIAL" is no good since the initial values of
   both "flow_id" and "sequence_no" vary from flow to flow.  "INITIAL"
   is only applicable where the initial value of a field is fixed, as is
   often the case with control fields.
```

B.6. Multiple Packet Formats

   To communicate initial values for the sequence number and flow ID
   fields correctly, and to take advantage of the fact that the flag
   bits are usually the same as in the previous header, we need to
   depart from the single format encoding we are currently using and
   instead use multiple formats. Here, we have expressed the encodings
   for two of the fields in the uncompressed format, since they will
   always be true for uncompressed headers of that format. The
   remaining fields, whose encoding method may depend on exactly how the
   header is being compressed, have their encodings specified in the
   compressed formats.

```
     eg_header
     {
       UNCOMPRESSED {
         version_no    =:= uncompressed_value(2, 1) [ 2 ];
         type                                       [ 2 ];
         flow_id                                    [ 4 ];
         sequence_no                                [ 4 ];
         abc_flag_bits                              [ 3 ];
         reserved_flag =:= uncompressed_value(1, 0) [ 1 ];
       }


       COMPRESSED irregular_format {
         discriminator =:= '0'         [ 1 ];
         version_no                    [ 0 ];
         type          =:= irregular(2) [ 2 ];
         flow_id       =:= irregular(4) [ 4 ];
         sequence_no   =:= irregular(4) [ 4 ];
         abc_flag_bits =:= irregular(3) [ 3 ];
         reserved_flag                 [ 0 ];
       }

       COMPRESSED compressed_format {
         discriminator =:= '1'         [ 1 ];
         version_no                    [ 0 ];
         type          =:= irregular(2) [ 2 ];
         flow_id       =:= static      [ 0 ];
         sequence_no   =:= lsb(2, -3)  [ 2 ];
```

```
      abc_flag_bits =:= static       [ 0 ];
      reserved_flag                  [ 0 ];
    }
  }
```

Note that we have added a discriminator field, so that the
decompressor can tell which format has been used by the compressor.
The format with a "static" flow ID and "lsb" encoded sequence number
is now 5 bits long. Note that despite having to add the
discriminator field, this format is still the same size as the
original incorrect "obvious" format because it takes advantage of the
fact that the abc flag bits rarely change.

However, the original "basic" format has also grown by one bit due to
the addition of the discriminator ("irregular_format"). An important
consideration when creating multiple formats is whether each format
occurs frequently enough that the average compressed header length is
shorter as a result of its usage. For example, if in fact the flag
bits always changed between packets, the "compressed_format" encoding
could never be used; all we would have achieved is lengthening the
"basic" format by one bit.

Using the above notation, we now get:

```
  Uncompressed header: 0101000100010000
  Compressed header:   00100010001000


  Uncompressed header: 0101000101000000
  Compressed header:   10100 ; 00100010100000


  Uncompressed header: 0110000101110000
  Compressed header:   11011 ; 01000010111000
```

The first header in the stream is compressed the same way as before,
except that it now has the extra 1-bit discriminator at the start
(0). When a second header arrives with the same flow ID as the first
and its sequence number three higher, it can be compressed in two
possible ways: either by using "compressed_format" or, in the same
way as previously, by using "irregular_format".

Note that we show all theoretically possible encodings of a header as
defined by the ROHC-FN specification, separated by semi-colons.
Either of the above encodings for each header could be produced by a
valid implementation, although a good implementation would always aim
to pick the encoding that leads to the best compression. A good
implementation would also take robustness into account and therefore

probably wouldn't assume on the second packet that the decompressor
had available the context necessary to decompress the shorter
"compressed_format" form.

Finally, note that the fields whose encoding methods are specified in
the uncompressed format have zero length when compressed. This means
their position in the compressed format is not significant. In this
case, there is no need to notate them when defining the compressed
formats. In the next part of the example we will see that they have
been removed from the compressed formats altogether.

B.7. Variable Length Discriminators

Suppose we do some analysis on flows of our example protocol and
discover that whilst it is usual for successive packets to have the
same flags, on the occasions when they don't, the packet is almost
always a "flags set" packet in which all three of the abc flags are
set. To encode the flow more efficiently a format needs to be
written to reflect this.

This now gives a total of three formats, which means we need three
discriminators to differentiate between them. The obvious solution
here is to increase the number of bits in the discriminator from one
to two and use discriminators 00, 01, and 10 for example. However we
can do slightly better than this.

Any uniquely identifiable discriminator will suffice, so we can use
00, 01, and 1. If the discriminator starts with 1, that's the whole
thing. If it starts with 0, the decompressor knows it has to check
one more bit to determine the kind of format.

Note that care must be taken when using variable length
discriminators. For example, it would be erroneous to use 0, 01, and
10 as discriminators since after reading an initial 0, the
decompressor would have no way of knowing if the next bit was a
second bit of discriminator, or the first bit of the next field in
the format. However, 0, 10, and 11 would be correct, as the first
bit again indicates whether or not there are further discriminator
bits to follow.

This gives us the following:

```
eg_header
{
  UNCOMPRESSED {
    version_no    =:= uncompressed_value(2, 1) [ 2 ];
    type                                       [ 2 ];
    flow_id                                    [ 4 ];
    sequence_no                                [ 4 ];
    abc_flag_bits                              [ 3 ];
    reserved_flag =:= uncompressed_value(1, 0) [ 1 ];
  }

  COMPRESSED irregular_format {
    discriminator =:= '00'        [ 2 ];
    type          =:= irregular(2) [ 2 ];
    flow_id       =:= irregular(4) [ 4 ];
    sequence_no   =:= irregular(4) [ 4 ];
    abc_flag_bits =:= irregular(3) [ 3 ];
  }

  COMPRESSED flags_set {
    discriminator =:= '01'                     [ 2 ];
    type          =:= irregular(2)             [ 2 ];
    flow_id       =:= static                   [ 0 ];
    sequence_no   =:= lsb(2, -3)               [ 2 ];
    abc_flag_bits =:= uncompressed_value(3, 7) [ 0 ];
  }

  COMPRESSED flags_static {
    discriminator =:= '1'         [ 1 ];
    type          =:= irregular(2) [ 2 ];
    flow_id       =:= static      [ 0 ];
    sequence_no   =:= lsb(2, -3)  [ 2 ];
    abc_flag_bits =:= static      [ 0 ];
  }
}
```

Here is some example output:

```
Uncompressed header: 0101000100010000
Compressed header:   000100010001000


Uncompressed header: 0101000101000000
Compressed header:   10100 ; 000100010100000
```

```
   Uncompressed header: 0110000101110000
   Compressed header:   11011 ; 001000010111000


   Uncompressed header: 0111000110101110
   Compressed header:   011110 ; 001100011010111
```

Here we have a very similar sequence to last time, except that there
is now an extra message on the end that has the flag bits set. The
encoding for the first message in the stream is now one bit larger,
the encoding for the next two messages is the same as before, since
that format has not grown; thanks to the use of variable length
discriminators. Finally, the packet that comes through with all the
flag bits set can be encoded in just six bits, only one bit more than
the most common format. Without the extra format, this last packet
would have to be encoded using the longest format and would have
taken up 14 bits.

B.8. Default Encoding

Some of the common encoding methods used so far have been "factored
out" into the definition of the uncompressed format, meaning that
they don't need to be defined for every compressed format. However,
there is still some redundancy in the notation. For a number of
fields, the same encoding method is used several times in different
formats (though not necessarily in all of them), but the field
encoding is redefined explicitly each time. If the encoding for any
of these fields changed in the future, then every format that uses
that encoding would have to be modified to reflect this change.

This problem can be avoided by specifying default encoding methods
for these fields. Doing so can also lead to a more concisely notated
profile:

```
     eg_header
     {
       UNCOMPRESSED {
         version_no    =:= uncompressed_value(2, 1) [ 2 ];
         type                                       [ 2 ];
         flow_id                                     [ 4 ];
         sequence_no                                 [ 4 ];
         abc_flag_bits                               [ 3 ];
         reserved_flag =:= uncompressed_value(1, 0) [ 1 ];
       }

       DEFAULT {
         type          =:= irregular(2);
         flow_id       =:= static;
```

```
        sequence_no   =:= lsb(2, -3);
      }

      COMPRESSED irregular_format {
        discriminator =:= '00'        [ 2 ];
        type                          [ 2 ]; // Uses default
        flow_id       =:= irregular(4) [ 4 ]; // Overrides default
        sequence_no   =:= irregular(4) [ 4 ]; // Overrides default
        abc_flag_bits =:= irregular(3) [ 3 ];
      }

      COMPRESSED flags_set {
        discriminator =:= '01' [ 2 ];
        type                   [ 2 ]; // Uses default
        sequence_no            [ 2 ]; // Uses default
        abc_flag_bits =:= uncompressed_value(3, 7);
      }

      COMPRESSED flags_static {
        discriminator =:= '1' [ 1 ];
        type                  [ 2 ]; // Uses default
        sequence_no           [ 2 ]; // Uses default
        abc_flag_bits =:= static;
      }
    }
```

The above profile behaves in exactly the same way as the one notated
previously, since it has the same meaning. Note that the purpose
behind the different formats becomes clearer with the default
encoding methods factored out: all that remains are the encodings
that are specific to each format. Note also that default encoding
methods that compress down to zero bits have become completely
implicit. For example the compressed formats using the default
encoding for "flow_id" don't mention it (the default is "static"
encoding that compresses to zero bits).

B.9. Control Fields

One inefficiency in the compression scheme we have produced thus far
is that it uses two bits to provide the "lsb" encoded sequence number
with robustness for the loss of just one packet. In theory, only one
bit should be needed. The root of the problem is the unusual
sequence number that the protocol uses -- it counts up in increments
of three. In order to encode it at maximum efficiency we need to
translate this into a field that increments by one each time. We do
this using a control field.

A control field is extra data that is communicated in the compressed
format, but which is not a direct encoding of part of the
uncompressed header. Control fields can be used to communicate extra
information in the compressed format, that allows other fields to be
compressed more efficiently.

The control field that we introduce scales the sequence number down
by a factor of three. Instead of encoding the original sequence
number in the compressed packet, we encode the scaled sequence
number, allowing us to have robustness to the loss of one packet by
using just one bit of "lsb" encoding:

```
eg_header
{
  UNCOMPRESSED {
    version_no    =:= uncompressed_value(2, 1) [ 2 ];
    type                                       [ 2 ];
    flow_id                                    [ 4 ];
    sequence_no                                [ 4 ];
    abc_flag_bits                              [ 3 ];
    reserved_flag =:= uncompressed_value(1, 0) [ 1 ];
  }

  CONTROL {
    // need modulo maths to calculate scaling correctly,
    // due to 4 bit wrap around
    scaled_seq_no   [ 4 ];
    ENFORCE(sequence_no.UVALUE
            == (scaled_seq_no.UVALUE * 3) % 16);
  }

  DEFAULT {
    type          =:= irregular(2);
    flow_id       =:= static;
    scaled_seq_no =:= lsb(1, -1);
  }

  COMPRESSED irregular_format {
    discriminator =:= '00'         [ 2 ];
    type                           [ 2 ];
    flow_id       =:= irregular(4) [ 4 ];
    scaled_seq_no =:= irregular(4) [ 4 ]; // Overrides default
    abc_flag_bits =:= irregular(3) [ 3 ];
  }

  COMPRESSED flags_set {
    discriminator =:= '01' [ 2 ];
    type                   [ 2 ];
```

```
      scaled_seq_no           [ 1 ]; // Uses default
      abc_flag_bits =:= uncompressed_value(3, 7);
    }

    COMPRESSED flags_static {
      discriminator =:= '1' [ 1 ];
      type                    [ 2 ];
      scaled_seq_no           [ 1 ]; // Uses default
      abc_flag_bits =:= static;
    }
  }
```

Normally, the encoding method(s) used to encode a field specifies the
length of the field. In the above notation, since there is no
encoding method using "sequence_no" directly, its length needs to be
defined explicitly using an "ENFORCE" statement. This is done using
the abbreviated syntax, both for consistency and also for ease of
readability. Note that this is unusual: whereas the majority of
field length indications are redundant (and thus optional), this one
isn't. If it was removed from the above notation, the length of the
"sequence_no" field would be undefined.

Here is some example output:

     Uncompressed header: 0101000100010000
     Compressed header: 000100011011000

     Uncompressed header: 0101000101000000
     Compressed header: 1010 ; 000100011100000

     Uncompressed header: 0110000101110000
     Compressed header: 1101 ; 001000011101000

     Uncompressed header: 0111000110101110
     Compressed header: 01110 ; 001100011110111

In this form, we see that this gives us a saving of a further bit in
most packets. Assuming the bulk of a flow is made up of
"flags_static" headers, the mean size of the headers in a compressed
flow is now just over a quarter of their size in an uncompressed
flow.

B.10. Use of "ENFORCE" Statements as Conditionals

   Earlier, we created a new format "flags_set" to handle packets with
   all three of the flag bits set. As it happens, these three flags are
   always all set for "type 3" packets, and are never all set for other
   packet types (a "type 3" packet is one where the type field is set to
   three).

   This allows extra efficiency in encoding such packets. We know the
   type is three, so we don't need to encode the type field in the
   compressed header. The type field was previously encoded as
   "irregular(2)", which is two bits long. Removing this reduces the
   size of the "flags_set" format from five bits to three, making it the
   smallest format in the encoding method definition.

   In order to notate that the "flags_set" format should only be used
   for "type 3" headers, and the "flags_static" format only when the
   type isn't three, it is necessary to state these conditions inside
   each format. This can be done with an "ENFORCE" statement:

```
     eg_header
     {
       UNCOMPRESSED {
         version_no    =:= uncompressed_value(2, 1) [ 2 ];
         type                                       [ 2 ];
         flow_id                                    [ 4 ];
         sequence_no                                [ 4 ];
         abc_flag_bits                              [ 3 ];
         reserved_flag =:= uncompressed_value(1, 0) [ 1 ];
       }

       CONTROL {
         // need modulo maths to calculate scaling correctly,
         // due to 4 bit wrap around
         scaled_seq_no   [ 4 ];
         ENFORCE(sequence_no.UVALUE
                 == (scaled_seq_no.UVALUE * 3) % 16);
       }

       DEFAULT {
         type          =:= irregular(2);
         scaled_seq_no =:= lsb(1, -1);
         flow_id       =:= static;
       }

       COMPRESSED irregular_format {
         discriminator =:= '00'         [ 2 ];
         type                           [ 2 ];
```

```
      flow_id       =:= irregular(4) [ 4 ];
      scaled_seq_no =:= irregular(4) [ 4 ];
      abc_flag_bits =:= irregular(3) [ 3 ];
    }

    COMPRESSED flags_set {
      ENFORCE(type.UVALUE == 3); // redundant condition
      discriminator =:= '01'                     [ 2 ];
      type          =:= uncompressed_value(2, 3)  [ 0 ];
      scaled_seq_no                               [ 1 ];
      abc_flag_bits =:= uncompressed_value(3, 7)  [ 0 ];
    }

    COMPRESSED flags_static {
      ENFORCE(type.UVALUE != 3);
      discriminator =:= '1'    [ 1 ];
      type                     [ 2 ];
      scaled_seq_no            [ 1 ];
      abc_flag_bits =:= static [ 0 ];
    }
  }
```

The two "ENFORCE" statements in the last two formats act as "guards".
Guards prevent formats from being used under the wrong circumstances.
In fact, the "ENFORCE" statement in "flags_set" is redundant. The
condition it guards for is already enforced by the new encoding
method used for the "type" field. The encoding method
"uncompressed_value(2,3)" binds the "UVALUE" attribute to three.
This is exactly what the "ENFORCE" statement does, so it can be
removed without any change in meaning. The "uncompressed_value"
encoding method on the other hand is not redundant. It specifies
other bindings on the type field in addition to the one that the
"ENFORCE" statement specifies. Therefore it would not be possible to
remove the encoding method and leave just the "ENFORCE" statement.

Note that a guard is solely preventative. A guard can never force a
format to be chosen by the compressor. A format can only be
guaranteed to be chosen in a given situation if there are no other
formats that can be used instead. This is demonstrated in the
example output below. The compressor can still choose the
"irregular" format if it wishes:

  Uncompressed header: 0101000100010000
  Compressed header: 000100011011000

  Uncompressed header: 0101000101000000
  Compressed header: 1010 ; 000100011100000

```
Uncompressed header: 0110000101110000
Compressed header:   1101 ; 001000011101000


Uncompressed header: 0111000110101110
Compressed header:   010 ; 001100011110111
```

This saves just two extra bits (a 7% saving) in the example flow.

Authors' Addresses

Robert Finking
Siemens/Roke Manor Research
Old Salisbury Lane
Romsey, Hampshire SO51 0ZN
UK

Phone: +44 (0)1794 833189
EMail: robert.finking@roke.co.uk
URI: http://www.roke.co.uk

Ghyslain Pelletier
Ericsson
Box 920
Lulea SE-971 28
Sweden

Phone: +46 (0) 8 404 29 43
EMail: ghyslain.pelletier@ericsson.com

Intellectual Property

   The IETF takes no position regarding the validity or scope of any
   Intellectual Property Rights or other rights that might be claimed to
   pertain to the implementation or use of the technology described in
   this document or the extent to which any license under such rights
   might or might not be available; nor does it represent that it has
   made any independent effort to identify any such rights. Information
   on the procedures with respect to rights in RFC documents can be
   found in BCP 78 and BCP 79.

   Copies of IPR disclosures made to the IETF Secretariat and any
   assurances of licenses to be made available, or the result of an
   attempt made to obtain a general license or permission for the use of
   such proprietary rights by implementers or users of this
   specification can be obtained from the IETF on-line IPR repository at
   http://www.ietf.org/ipr.

   The IETF invites any interested party to bring to its attention any
   copyrights, patents or patent applications, or other proprietary
   rights that may cover technology that may be required to implement
   this standard. Please address the information to the IETF at
   ietf-ipr@ietf.org.