Weave TLV Schema

Revision 1.0
2020/05/03

Introduction

A Weave TLV schema (or more simply, a TLV Schema) provides a simple textual language for describing data formats and other constructs typically used in Weave-based applications. Its primary use is to describe the structure of data encoded in Weave TLV format, a compact binary data format targeting constrained IoT devices. It can also describe certain other higher level constructs common in Weave applications, such as Weave profiles, message types and status codes. Emphasis has been placed on the inherent readability of the language, making it well suited for use in protocol documentation and formal specifications.

Relationship to Weave Data Language (WDL)

The Weave Data Language is an object modeling language that is used to describe the features of a communicating system. Whereas Weave TLV schemas describe relatively low-level constructs—things such as message formats and data encodings—WDL's purpose is to describe higher-level system interface concepts. These include such things as the externally visible properties of a system, the commands that can be directed at that system, and the events spontaneously produced by the system. WDL is also capable of modelling distributed functionality, where the services of a single logical system are spread out over multiple physical entities.

The concepts modelled by WDL are abstract in nature in that they do not dictate a particular over-the-wire format or message formulation. In practice, this means that a message containing a WDL property or command can be encoded in a variety of forms, including Weave TLV or Google protocol buffers[1]. Accordingly, it is possible (at least conceptually) to construct a Weave TLV schema for every communication implied by a WDL construct. Such a schema would describe how the abstract data implied by the WDL model is concretely encoded in Weave TLV format.

Another distinction between WDL and Weave TLV schemas is that the latter is more expressive in terms of the TLV constructs it can describe. A Weave TLV schema can describe the full range of possible Weave TLV encodings. WDL, on the other hand, is forced to accommodate the limitations of all possible formats in which it can be represented. Thus, there are particular patterns of TLV encodings (often those that are the most compact) which cannot be achieved in WDL.

Ultimately, the Weave Data Language and Weave TLV schemas exist to serve different purposes. WDL is a high-level language for the abstract modelling of communicating systems,

[1] Note that WDL borrows its syntax from the Google protocol buffer language, but both extends and constrains it to serve as an abstract modelling language.

while Weave TLV schemas are low-level specifications of over-the-wire data encodings that use Weave TLV form.

Basic Structure

A Weave TLV schema takes the form of a series of definitions. Each definition describes some construct, such as a data structure, that is used by a Weave application. For example, the following schema defines a data type that can be used to represent a temperature sample:

Example

```
/** Temperature sample structure */
temperature-sample => STRUCTURE
{
  timestamp [1] : UNSIGNED INTEGER [range 32bits],
  temperature [2] : FLOAT,
}
```

Each definition has an associated human readable name (temperature-sample in this case) separated from the definition with a **=>** symbol. As a mnemonic device, it is useful to read the **=>** symbol as "is a". For example, *temperature-sample is a structure containing a timestamp and temperature value*.

A Weave TLV schema can contain multiple definitions. Typically, TLV schemas are organized into files containing related sets of definitions. The order of definitions within a file is unimportant, and complex schemas can be broken up into multiple files for convenience.

Keywords

TLV schemas employ various keywords when describing a schema construct. These keywords (e.g. `STRUCTURE`, `INTEGER`, and `range`) are an inherent part of the schema language. Keywords in TLV schemas are always case-insensitive. However, by convention, keywords associated with types and other high-level constructs are capitalized for emphasis in text-only contexts.

Naming

Each definition in a TLV schema assigns a human-readable name to the construct being defined. This name serves both as a descriptive title as well as a means to refer to the construct from elsewhere in the schema.

Names in TLV schemas are limited to ASCII alphanumeric characters, plus dash (`-`) and underbar (`_`). Additionally, all names must begin with either an alphabetic character or an underbar. In general, any name conforming to these rules can be used. In some cases, however, a chosen name may collide with a keyword used by the schema language. In this situation, the name may be enclosed in double quotes (`"`) to avoid ambiguity.

Namespaces

The name assigned to a schema construct must be unique relative to all other named constructs in the same scope. To facilitate this, TLV schemas support a namespacing mechanism similar to that provided in languages like C++.

The names of constructs defined within a namespace definition are only required to be unique within the given namespace. Namespaces themselves can be nested to any depth.

Constructs defined in other namespaces can be referenced using a name that gives the enclosing namespaces, plus the construct name, each separated by dots (`.`). Such a multi-part name is called a *scoped name*. For example:

Example

```
namespace a
{
  x => STRING,
  other-x => b.x
}

namespace b
{
  x => INTEGER
}
```

Qualifiers

Constructs within a Weave TLV schema can be annotated with additional information using a *qualifier*. Qualifiers appear within square brackets (`[...]`) immediately following the construct they affect. A qualifier takes the form of a keyword optionally followed by an argument.

Often qualifiers are used to place restrictions on the form or range of values that a construct can assume. For example a `length` qualifier can be used to constrain the length of a STRING type:

Example

```
international-standard-book-number => STRING [length 13]
```

Multiple qualifiers can appear within the square brackets, separated by commas.

Tagging

Being designed for use over low-bandwidth networks, Weave TLV eschews the use of textual names to identify values in encodings, opting for the encoding of numeric tags instead. In a TLV schema, tag numbers appear as qualifiers attached to a particular named construct, such as a field within a structure. This association reflects the tag's role as an alias for the textual name in the corresponding encoding.

Weave TLV supports two kinds of tags: a *profile-specific* tag that is globally unique (e.g. `[comfort-sensing:1]`), and a *context-specific* tag that is unique within the context in which it is used (e.g. `[1]`).

Further details on tags and their forms can be found in the section describing the `tag` qualifier.

Example

```
temperature-sample [comfort-sensing:1] => STRUCTURE
{
  timestamp [1] : UNSIGNED INTEGER [range 32bits],
  temperature [2] : FLOAT,
}
```

Definitions

A Weave TLV schema consists of a set of one or more definitions. The types of definitions that can appear within a schema are:

- type definitions
- namespace definitions
- PROFILE definitions
- VENDOR definitions
- MESSAGE definitions
- STATUS CODE definitions

Type Definitions

Syntax

```
type-name [ qualifiers... ]opt => type-or-ref

type-or-ref:
   type
   type-ref

type:
   ANY
   ARRAY
   ARRAY OF
   BOOLEAN
   BYTE STRING
   CHOICE OF
   FIELD GROUP
   FLOAT
   INTEGER
   LIST
   LIST OF
   NULL
   SIGNED INTEGER
   STRING
   STRUCTURE
   UNSIGNED INTEGER

type-ref:
   type-name
   scoped-type-name
```

Allowed Qualifiers: `tag`

A type definition associates a name with a schema construct representing a TLV type or pseudo-type. The given name serves as a descriptive title for the type, as well as a means to refer to the type from elsewhere in the schema.

Type definitions are often used to describe TLV types that appear directly in some form of communication. For example, a type definition may define the structure of data carried within the payload of a Weave message. Some type definitions may be used to define general purpose TLV constructs which are then employed in the definitions of other types.

The type associated with a type definition can be any one of the available TLV types or pseudo-types. Alternatively, a type definition can contain a scoped name referring to another

type definition appearing elsewhere in the schema. This form is referred to as a *type reference*. The ordering of type definitions and type references within a schema is unimportant, implying that a type reference can refer to a type that is defined later in the schema file, or even in a different schema file.

A `tag` qualifier may be applied to the name within a type definition to associate a default tag with that name. The default tag will be used in an encoding of the type whenever an explicit tag has not been given.

For more information on default tags see the description of the `tag` qualifier.

Namespace Definitions

Syntax

```
namespace ns-name { ns-scoped-def... }

ns-scoped-def:
   type-def
   profile-def
   namespace-def

ns-name:
   name
   scoped-name
```

`namespace` introduces a new naming scope. Definitions that appear within the braces of a namespace definition are scoped to that namespace, such that their names need only be unique within the bounds of the enclosing scope.

In general, three forms of definitions may appear within a namespace: type definitions, profile definitions and further namespace definitions. Namespace definitions can be nested to any level. Profile definitions, however, are restricted such that they cannot be nested (see PROFILE for further details). Thus a namespace can only contain a profile definition if the namespace itself is not located, at any level, within another profile definition.

The name used in a namespace definition can be either a simple name, such as `a`, or a scoped name, such as `a.b.c`. When a scoped name is used, the effect is exactly as if multiple nested namespaces had been declared, each named after a part of the scoped name.

It is legal to have multiple namespace definitions, each with the same name, defined within the same scope. The effect is as if there was only a single namespace definition containing a union of the enclosed definitions. Thus, a namespace definition with the same name as a preceding

definition can be seen as a kind of continuation of the earlier one. Such continuations can occur within a single schema file, or across files.

Example

```
namespace hvac-types
{
  set-point-temp => FLOAT [ range 0..50 ]

  set-point => STRUCTURE
  {
    time-of-day [0] : UNSIGNED INTEGER,
    target-temp [1] : set-point-temp
  }
}

namespace weave.profiles.thermostat
{
  thermostat-config => STRUCTURE
  {
    set-points [0] : ARRAY OF hvac-types.set-point
    ...
  }
}
```

PROFILE Definitions

Syntax

```
name => PROFILE [ id profile-id ] { profile-scoped-def... }

profile-id:
   uint-value
   vendor-id:profile-num
   vendor-name:profile-num

profile-scoped-def:
   type-def
   namespace-def
   message-def
   status-code-def
```

Allowed Qualifiers: `id`

`PROFILE` defines a Weave profile. A Weave profile is a group of logically related Weave constructs—such as TLV types, message types, and status codes—that together serve a

common purpose. Each Weave profile has an associated unsigned 32-bit number, called a *profile id*, that uniquely identifies the profile among all other profiles. Weave profile ids are composed of two 16-bit fields: a *Weave vendor id* (in the high 16-bits), which identifies the company or organization that created the profile, and a *profile number* (in the low 16-bits), assigned by that vendor to uniquely identify the profile.

Similar to a namespace definition, a `PROFILE` definition introduces a new naming scope in which further definitions may appear. The names of definitions appearing within the braces of a `PROFILE` are scoped in exactly the same way as if they had appeared within a `namespace` definition. Likewise, constructs outside the `PROFILE` definition can refer to definitions within the profile by using a scoped name that includes the profile name.

`PROFILE` definitions may appear at the global naming scope, or within a namespace definition. However, `PROFILE` definitions *may not* be nested within other `PROFILE` definitions at any depth.

Every `PROFILE` definition must include an `id` qualifier giving the id of the profile. The value of the `id` qualifier can be specified either as a single integer value, in hex or decimal, or as a tuple of vendor id and profile number integers, separated by a colon. If a VENDOR definition exists elsewhere in the schema, the vendor portion of the profile id can be expressed as a name, rather than as an integer value.

The id given in a `PROFILE` definition must be unique relative to all other `PROFILE` definitions in a schema. However, it is legal to have multiple `PROFILE` definitions with the same profile id, provided that they also have the same name and appear within the same naming scope. The effect of this is as if there were only a single `PROFILE` definition containing a union of the enclosed definitions. This makes it possible to break up a `PROFILE` definition across multiple schema files.

Example

```
namespace weave.profiles {

device-description => PROFILE [ id common:0x000E ]
{
  device-descriptor => STRUCTURE
  {
    vendor-id [0] : UNSIGNED INTEGER,
    product-id [1] : UNSIGNED INTEGER,
    product-revision [2,opt] : UNSIGNED INTEGER,

    ...
  }
}

}
```

VENDOR Definitions

Syntax

```
name => VENDOR [ id vendor-id ]

vendor-id:
   uint-value
```

Allowed Qualifiers: `id`

`VENDOR` associates a name with a Weave vendor id. Weave vendor ids are 16-bit unsigned integers that uniquely identify an organization or company that participates in the Weave ecosystem. Vendor ids are used to scope other identifiers (e.g. profile ids) such that organizations can independently mint these identifiers without fear of collision.

In a TLV schema that includes a `VENDOR` definition, the vendor name may be used elsewhere in the schema as a stand-in for the associated vendor id. One such place where a vendor name may appear is within the id qualifier of a `PROFILE` definition.

`VENDOR` definitions can only appear at the global name scope, implying they cannot be placed within the body of a namespace or `PROFILE` definition.

Both the name and id value used in a `VENDOR` definition must be unique across all such definitions. However, for convenience, a `VENDOR` definition may be repeated (in the same schema file or in different files) provided that the name and id are the same.

The special vendor id 0 represents the Weave “common” vendor. The common vendor is used to scope universal Weave constructs that are not owned or defined by any single organization. The common vendor is implicitly defined in all schemas, although it may be explicitly defined as well.

Example

```
common => VENDOR [ id 0 ]

nest => VENDOR [ id 0x235A ]

google => VENDOR [ id 0xE100 ]
```

MESSAGE Definitions

Syntax

```
name => MESSAGE [ id msg-type-num ]
name => MESSAGE [ id msg-type-num ] CONTAINING type
name => MESSAGE [ id msg-type-num ] CONTAINING NOTHING

msg-type-num:
   uint-value
```

Allowed Qualifiers: `id`

`MESSAGE` defines a Weave message. Weave messages provide a general-purpose framework for building application-level IoT protocols.

The meaning of a Weave message is conveyed by means of type information encoded in the message header. Message type information consists of a tuple of two integer values: a 32-bit *Weave profile id*, which is assigned to the Weave profile in which the message is defined; and an 8-bit *message type number*, which distinguishes the message from others in the same profile. (For more on Weave profile ids, see the section on PROFILE Definitions).

All `MESSAGE` definitions must appear immediately within the body of a PROFILE definition. The id of the enclosing profile determines the profile id for the message.

The message type number is specified by means of an `id` qualifier which must appear in the `MESSAGE` definition. The given message type number must be unique across all `MESSAGE` definitions within the same profile. The value of the `id` qualifier can be specified in either hex or decimal form.

A `MESSAGE` definition may include a *containing clause*, consisting of the keyword `CONTAINING` followed by a TLV type or pseudo-type. If present, the containing clause defines the syntax of the data conveyed within the message payload (the data being the TLV encoding of the specified type).

Omitting the containing clause indicates that the syntax of the message payload is defined outside the purview of TLV schemas. This is often used for messages with bespoke payloads, or whose payloads are defined by an external standard. Typically, the structure of the payload is described in the prose documentation for the message.

Within a containing clause, the special keyword `NOTHING` can be used in place of a type to define a message whose payload must always be empty.

Example

```
device-description => PROFILE [ id common:0x000E ]
{
  identify-request => MESSAGE [ id 1 ]

  identify-response => MESSAGE [ id 2 ] CONTAINING device-descriptor

  device-descriptor => STRUCTURE
  {
    ...
  }
}
```

STATUS CODE Definitions

Syntax

```
name => STATUS CODE [ id status-code-num ]

status-code-num:
   uint-value
```

Allowed Qualifiers: `id`

`STATUS CODE` defines a Weave status code. A Weave status code is a globally unique value used to signal the result of an operation. Status codes are typically conveyed in Weave Status Report messages (defined within the Weave common profile), but can be used in other contexts as well. The Weave common profile defines a set of generally useful status codes, however application developers are free to define other codes as needed within their own Weave profiles.

A Weave status code consists of a tuple of two unsigned integer values: a 32-bit *Weave profile id*, which is assigned to the Weave profile in which the status code is defined; and a 16-bit *status code number*, which distinguishes the code from others in the same profile. (For more on Weave profile ids, see the section on PROFILE Definitions).

The status code number is specified by means of an `id` qualifier which must appear in the `STATUS CODE` definition. The given number must be unique across all `STATUS CODE` definitions within the same profile. The value of the `id` qualifier can be specified in either hex or decimal form.

All `STATUS CODE` definitions must appear immediately within the body of a PROFILE definition. The id of the enclosing profile determines the profile id for the status code.

Example

```
network-provisioning => PROFILE [ id common:0x0003 ]
{
  unknown-network                => STATUS CODE [ id 1 ]
  too-many-networks              => STATUS CODE [ id 2 ]
  invalid-network-configuration => STATUS CODE [ id 3 ]
  unsupported-network-type       => STATUS CODE [ id 4 ]
}
```

Types

The Weave TLV format supports 10 fundamental types: integers (signed and unsigned), floats, booleans, strings, byte strings, structures, arrays, lists and nulls. Accordingly, a Weave TLV schema can use one of the following type constructs to constrain an encoding to be one of these fundamental types.

ARRAY / ARRAY OF

Syntax

```
ARRAY [ qualifiers... ]opt OF type-or-ref          → uniform array
ARRAY [ qualifiers... ]opt { type-pattern... }     → pattern array

type-pattern:
  type-or-ref quantifieropt                        → unnamed item
  identifier : type-or-ref quantifieropt           → named item

quantifier:
  *                                                → zero or more
  +                                                → one or more
  { count }                                        → exactly count
  { min..max }                                     → between min and max
  { min.. }                                        → at least min
```

Allowed Qualifiers: `length`, `nullable`

`ARRAY` and `ARRAY OF` declare an element that is encoded as a TLV Array type. TLV Arrays are sequences of TLV elements all of which are untagged (i.e. tagged with the 'anonymous' tag). Although `ARRAY` and `ARRAY OF` both declare the same fundamental type, they differ based on how the types of their items are expressed.

`ARRAY OF` declares an array where all the items in the array are of the same fundamental type, or taken from the same set of possible types. This form of array is called a *uniform array*, and is generally used to represent ordered collections of values.

`ARRAY` declares an array where the types of the array items follow a particular pattern. In this form, known as a *pattern array*, the allowed type for an item depends on its position in the array. The overall pattern of types allowed in the array is declared using a schema construct called a linear type pattern, which is similar to a regular expression (see below). Pattern arrays are typically used to represent vectors, tuples or paths.

Note that a `FIELD GROUP` cannot be used as an item type within an `ARRAY` or `ARRAY OF`.

A `length` qualifier on an array can be used to constraint the minimum and maximum number of items in the array. For a pattern array, the given length constraint must be consistent with (i.e. fall within) the minimum and maximum number of items implied by the type pattern. In cases where the `length` qualifier places a narrower constraint on the length of an array than that implied by the type pattern, the `length` qualifier constraint takes precedence.

A `nullable` qualifier can be used to indicate that a TLV Null may be encoded in place of the `ARRAY`/`ARRAY OF`. Note that an array that has been replaced by a Null is distinct in terms of its encoding from an array that has no items.

Linear Type Patterns

A linear type pattern describes the sequence of TLV types that may appear in a TLV Array or List element. In its simplest form, a linear type pattern is a list of type definitions, or references to defined types, where each item constrains the TLV type that appears at the corresponding position in the collection. The type pattern is always anchored at the start of the collection, with the first type constraining the first item in the collection. Any type or pseudo-type may appear within a linear type pattern with the exception of a `FIELD GROUP`.

More complex type patterns can be created by using a *quantifier*. Quantifiers appear after a type in a type pattern and specify the number of times the associated type may appear at that position in the collection. Quantifiers borrow common regular expression notation to denote repetition, with ***** meaning zero or more, **+** meaning one or more, and **{ }** expressing specific counts.

Using quantifiers, one can express complex sequences of types, including some that require arbitrary look-ahead to match. Schema authors should be aware, however, that code generators or dynamic schema checkers may not have the capability of enforcing such schemas.

Item Names

Items or groups of items in a pattern array can be given textual names. These names do not affect the encoding of the array, but serve as user documentation, or as input to code generation tools. Item names within a pattern array must be unique.

Per the rules for encoding TLV arrays, array items cannot have tags. Thus the tag qualifier cannot be applied to an item name with a pattern array.

Example

```
supported-country-codes => ARRAY [len 0..10] OF STRING [len 2]

weather-tuple => ARRAY
{
  timestamp                : UNSIGNED INTEGER [range 32bits],
  temperature              : FLOAT,
  relative-humidity        : UNSIGNED INTEGER [range 0..100],
  precipitation            : UNSIGNED INTEGER [range 0..100],
}

named-vector => ARRAY
{
  name                     : STRING,
                             FLOAT *,
}
```

BOOLEAN

Syntax

```
BOOLEAN [ qualifiers... ]opt
```

Allowed Qualifiers: `nullable`

`BOOLEAN` declares an element that must be encoded as a TLV boolean value; specifically, a TLV True or TLV False. If the `nullable` qualifier is given, a TLV Null can be encoded in its place.

Example

```
pathlight-enabled => BOOLEAN
```

BYTE STRING

Syntax

```
BYTE STRING [ qualifiers... ]opt
```

Allowed Qualifiers: `length`, `nullable`

`BYTE STRING` declares an element that is encoded as a TLV Byte String. TLV Byte Strings convey arbitrary-length binary data. If the `nullable` qualifier is given, a TLV Null can be encoded in place of the byte string.

The minimum and maximum number of bytes can be constrained using the `length` qualifier.

Example

```
mac-address-802-15-4 => BYTE STRING [ length 8 ]
```

FLOAT

Syntax

```
FLOAT [ qualifiers... ]opt
```

Allowed Qualifiers: `range`, `nullable`

`FLOAT` declares an element that is encoded as a TLV Float type. If the `nullable` qualifier is given, a TLV Null can be encoded in place of the number.

The allowed range of values can be constrained using the `range` qualifier. If omitted, the value is constrained to fit within 64-bits (the maximum size of a TLV floating point number).

Example

```
set-point-temp => FLOAT [ range 0..50 ]
```

INTEGER / SIGNED INTEGER / UNSIGNED INTEGER

Syntax

```
INTEGER [ qualifiers... ]opt { enums... }opt
SIGNED INTEGER [ qualifiers... ]opt { enums... }opt
UNSIGNED INTEGER [ qualifiers... ]opt { enums... }opt

enum:
   identifier = int-value
```

Allowed Qualifiers: `range`, `nullable`

`INTEGER` and `SIGNED INTEGER` declare an element that must be encoded as a TLV Signed Integer. Correspondingly, `UNSIGNED INTEGER` declares a TLV element that must be

encoded as a TLV Unsigned Integer. If the `nullable` qualifier is given, a TLV Null can be encoded in place of the integer.

The allowed range of values can be constrained using the `range` qualifier. If omitted, the value is constrained to fit within 64-bits (the maximum TLV integer size).

An `INTEGER` definition can include a set of enumerated values, each of which associates a textual name with a constant integer value. Each value must conform to the allowed range of values for the `INTEGER` definition as given by its sign and any `range` qualifier.

Note that presence of enumerated values does not restrict senders to only encoding those values. Rather, enumerations merely give symbolic names to particular noteworthy values.

Example

```
temp-sensor-value => INTEGER [ range -100..100 ]

event-counter => UNSIGNED INTEGER [ range 32bits ]
```

LIST / LIST OF

Syntax

```
LIST [ qualifiers... ]opt OF type-or-ref          → uniform list
LIST [ qualifiers... ]opt { type-pattern... }     → pattern list

type-pattern:
  type-or-ref quantifieropt                       → unnamed item
  identifier [ qualifiers... ]opt :  type-or-ref quantifieropt
                                                  → named item

quantifier:
  *                                               → zero or more
  +                                               → one or more
  { count }                                       → exactly count
  { min..max }                                    → between min and max
  { min.. }                                       → at least min
```

Allowed Qualifiers: `length`, `nullable`

Allowed Qualifiers:

- **LIST:** `length`, `nullable`
- *Element name identifier*: `tag`

`LIST` and `LIST OF` declare an element that is encoded as a TLV list type. TLV lists are sequences of TLV elements which may include tags. This makes lists distinct from TLV arrays, whose items cannot include tags. `LIST` and `LIST OF` declare the same fundamental type, but differ based on how the allowed types of their items are expressed.

> *NOTE: In early versions of the Weave TLV specification, TLV lists were referred to as "paths", reflecting their initial use in identifying nodes in hierarchical object trees.*

`LIST OF` declares a list where all the items in the list are of the same fundamental type, or taken from the same set of possible types. This form of list is called a *uniform list*. Uniform lists are generally used to represent ordered collections of values where the tags differentiate the semantic meaning of the value.

`LIST` declares a *pattern list* where the types of the items in the list follow a particular pattern. In this form, the allowed type(s) for an item depends on its position in the array. Pattern lists are typically used to represent path-like constructs.

The overall pattern of types allowed in a pattern list is declared using a schema construct called a linear type pattern. The syntax and interpretation of linear type patterns for pattern lists are the same as those for pattern arrays (see Linear Type Patterns).

The `length` qualifier can be used to constraint the minimum and maximum number of items in the list. For a pattern list, the given length constraint must be consistent with (i.e. fall within) the minimum and maximum number of items implied by the type pattern. In cases where the `length` qualifier places a narrower constraint on the length of a list than that implied by the type pattern, the `length` qualifier constraint takes precedence.

A `nullable` qualifier can be used to indicate that a TLV Null may be encoded in place of the `LIST`/`LIST OF`. Note that a list that has been replaced by a Null is distinct (in terms of its encoding) from a list that has no items.

Note that a `FIELD GROUP` cannot be used as an item type within a `LIST` or `LIST OF`.

Item Names

As with the `ARRAY` type, items or groups of items in a pattern list can be given textual names to distinguish their purposes. Item names within a pattern list must be unique.

Item Tags

Items within a pattern list can have a `tag` qualifier that specifies a particular tag value that must be encoded with the item. The specific tag can be profile-specific or context-specific, or the `anon` tag. The assigned tag values are not required to be unique among the items in a pattern list.

When no explicit tag qualifier is given (which is always the case for uniform lists) the items in a list automatically assume the default tag of their underlying types, if such a tag is provided. This can occur in two situations: 1) when the underlying type is a reference to a type definition that declares a default tag, and 2) when the underlying type is a `CHOICE OF` whose alternates declare default tags. See the section on Default Tags for further information.

If no tag qualifier is given, and no default tag is available, an encoder is allowed to encode list items with any tag of their choosing.

NULL

Syntax

```
NULL
```

Allowed Qualifiers: *none*

`NULL` declares an element that must be encoded as a TLV Null. There are no qualifiers that can be associated with a `NULL` type.

Example

```
general-device-id => CHOICE OF { STRING, UNSIGNED INTEGER, NULL }
```

STRING

Syntax

```
STRING [ qualifiers... ]opt
```

Allowed Qualifiers: `length`, `nullable`

`STRING` declares an element that is encoded as a TLV UTF-8 String. If the `nullable` qualifier is given, a TLV Null can be encoded in place of the string.

The minimum and maximum length of the string can be constrained using the `length` qualifier.

Example

```
user-name-field => STRING [ length 0..32 ]
```

STRUCTURE

Syntax

```
STRUCTURE [ qualifiers... ]opt { struct-members... }

struct-member:
   identifier [ qualifiers... ]opt : type-or-ref      → field definition
   includes type-ref                                  → field group include
```

Allowed Qualifiers:

- **STRUCTURE:** `extensible`, `tag-order`, `schema-order`, `any-order`, `nullable`
- *Field name identifier*: `tag`, `optional`

`STRUCTURE` declares an element that is encoded as a TLV Structure type. A TLV Structure is a collection of TLV elements, known as *fields*, each of which is identified by a TLV tag. The tags assigned to the fields of a TLV Structure must be unique as compared to all other fields within that structure. Untagged fields (i.e. those marked with an 'anonymous' tag) cannot appear within a TLV Structure.

A `STRUCTURE` definition declares the list of fields that may appear within the corresponding TLV Structure. Each field definition gives the type of the field, its tag, and an associated textual name. The field type can be either a fundamental type, a `CHOICE OF` pseudo-type, an `ANY` pseudo-type, or a reference to one of these types defined outside the `STRUCTURE` definition.

A `STRUCTURE` definition may also contain one or more `includes` statements. Each such statement identifies a `FIELD GROUP` pseudo-type whose fields are to be included within the TLV Structure as if they had been declared within the `STRUCTURE` definition itself (see Includes Statements and FIELD GROUPs below).

A `nullable` qualifier can be used to indicate that a TLV Null may be encoded in place of the `STRUCTURE`.

Field Names

Fields within a `STRUCTURE` are assigned textual names to distinguish them from one another. Each such name must be distinct from all other field names defined within the `STRUCTURE` or included via a `includes` statement. Fields names do not affect the encoding of the resultant TLV, but serve as either user documentation or input to code generation tools.

Field Tags

Per the rules of TLV, all fields within a TLV Structure must be encoded with a distinct TLV tag. Field tags are declared by placing a tag qualifier on the field name. Both profile-specific and context-specific tags are allowed on the fields in a `STRUCTURE` definition.

A field may omit a tag qualifier if the underlying type of the field provides a default tag. When this happens the default tag is used as the tag for the field. This can occur in two situations: 1) when the underlying type is a reference to a type definition that provides a default tag, and 2) when the underlying type is a `CHOICE OF` pseudo-type whose alternates provide default tags. See the section on Default Tags for further information.

The tags associated with included fields (see Includes Statements and FIELD GROUPs below) are inherited from the target `FIELD GROUP` definition.

All tags associated with the fields of a TLV Structure must be unique. This is true not only for tags declared directly within the `STRUCTURE` definition, but also for any tags associated with fields that are incorporated via an `includes` statement.

The `anon` tag cannot be used as the tag for a field within a `STRUCTURE` definition.

CHOICE OF Fields

A field within a `STRUCTURE` definition may be defined to be a `CHOICE OF` pseudo-type (either directly within the `STRUCTURE` definition or via a type reference). Over the wire, such a field is encoded as one of the alternate types given in the `CHOICE OF` definition. For example, the user-id field in the following structure can be encoded as either a TLV String or Unsigned Integer.

Example

```
user-information => STRUCTURE [ extensible ]
{
   user-id [1] : CHOICE OF
      {
         UNSIGNED INTEGER,
         STRING
      }
}
```

If a tag qualifier is given for a `CHOICE OF` field (e.g. [1] as shown above), that tag is used in the encoding of the field for all possible alternates. On the other hand, if a tag qualifier is *not* given, then the default tag associated with the selected `CHOICE OF` alternate is used in the encoding. For example, in the following structure, a context-tag of 1 will be encoded if the user-id field is an integer, or 2 if the field is a string.

Example

```
user-information => STRUCTURE [ extensible ]
{
   user-id : CHOICE OF
      {
         id [1] : UNSIGNED INTEGER,
         name [2] : STRING
      }
}
```

Note that, in all cases, the tag or tags associated with a `CHOICE OF` field must be unique within the context of the containing `STRUCTURE`.

Includes Statements and FIELD GROUPs

A `includes` statement may be used within a `STRUCTURE` definition to incorporate the fields of a `FIELD GROUP` type defined outside the `STRUCTURE`. The fields of the `FIELD GROUP` are included in the `STRUCTURE` as if they had been listed within the `STRUCTURE` definition itself. `FIELD GROUP`s and `includes` statements make it possible to factor common patterns of fields, and to reuse these patterns across different `STRUCTURE` definitions.

A particular `FIELD GROUP` may only be included once within a `STRUCTURE`.

The names assigned to fields within an included `FIELD GROUP` must be distinct with respect to all other fields contained within the enclosing `STRUCTURE`, whether defined directly within the `STRUCTURE` itself, or included from another `FIELD GROUP`. A

Likewise, tags assigned to fields within an included `FIELD GROUP` must be distinct with respect to all other fields within the enclosing `STRUCTURE`.

Optional Fields

The optional qualifier can be used to declare a field which can be omitted from the structure encoding under some circumstances. When used, the optional qualifier is applied to the name of a field within a `STRUCTURE` or `FIELD GROUP` definition.

The conditions under which an optional field can be omitted depend on the semantics of the structure. In some cases, fields may be omitted entirely at the discretion of the sender. In other cases, omission of a field may be contingent on the value present in another field. In all cases, prose documentation associated with the field definition should make clear the rules for when the field may be omitted.

Note that an optional field is distinct, both semantically and in terms of encoding, from a field whose type has been declared nullable. In the former case, it is legal to omit the field from the

structure encoding altogether, while in the latter case, the field must be encoded, but its value can be encoded as a TLV Null.

Optional fields are allowed within `FIELD GROUP`s and retain their optionality when included within `STRUCTURE`s.

Encoding Order

The `any-order`, `schema-order` and `tag-order` qualifiers can be used to specify a particular order for the encoding of fields within a TLV Structure.

The `schema-order` qualifier specifies that the fields of a structure must be encoded in the order given within the associated `STRUCTURE` definition. If the `STRUCTURE` definition contains one or more `includes` statements, the fields of the referenced `FIELD GROUP`s must be encoded in the order given in the respective `FIELD GROUP` definition, and at the position of the `includes` statement relative to other fields within the `STRUCTURE`.

The `tag-order` qualifier specifies that the fields of a structure must be encoded in increasing order of their tag value. When ordering tags in this way, context-specific tags implicitly appear before profile-specific tags, and profile-specific tags with lower numeric vendor ids implicitly appear before those with high vendor ids.

The `any-order` qualifier specifies that the encoder of a TLV structure is free to encode the fields of the structure in any desired order.

In the absence of an order qualifier, fields within TLV structure can generally be encoded in any order. However, the author of a `STRUCTURE` definition may choose to impose custom ordering constraints on some or all of the fields if so desired. Such constraints must be clearly described in the prose documentation for the schema.

Extensibility

An `extensible` qualifier can be used to declare that a structure can be extended at encoding time by the inclusion of fields not listed in the `STRUCTURE` definition. When a structure is extended in this way, any new fields included in the encoding must use tags that are distinct from any of those associated with defined or included fields.

Absent the `extensible` qualifier, a structure encoding may *not* include fields beyond those given in the `STRUCTURE` definition.

Pseudo-Types

Pseudo-types are type-like constructs that provide flexibility in schema definitions. Some pseudo-types, like `CHOICE OF` and `ANY`, allow for variance in the fundamental TLV types that make appear in an encoding. Others make it easier to reuse schema constructs in multiple contexts.

ANY

Syntax

```
ANY
```

Allowed Qualifiers: *none*

`ANY` declares an element that can be encoded as any fundamental TLV type. Note that `ANY` is not a fundamental TLV type itself, but rather a pseudo-type that identifies a range of possible encodings. An `ANY` type serves a shorthand for (and is exactly equivalent to) a `CHOICE OF` all possible fundamental types.

There are no qualifiers that can be associated with an `ANY` type.

Example

```
app-defined-metadata => ANY
```

CHOICE OF

Syntax

```
CHOICE [ qualifiers... ]opt OF { alternates... }

alternate:
  type-or-ref                                    → unnamed alternate
  identifier [ qualifiers... ]opt : type-or-ref  → named alternate
```

Allowed Qualifiers:

- **CHOICE OF:** `nullable`
- *alternate identifier*: `tag`

`CHOICE OF` declares an element that can be any of a set of TLV types. `CHOICE OF` is considered a pseudo-type, rather than a fundamental type, in that the `CHOICE OF` itself doesn't have a representation in the final TLV encoding.

The allowed TLV types for a `CHOICE OF`, known as *alternates*, are given in the body of the definition. An alternate can be any of the fundamental TLV types, an `ANY` pseudo-type, or another `CHOICE OF` definition (more on this below). Additionally, an alternate can be a type reference (in the form of a scoped type name) referring to a type defined outside of the `CHOICE OF` definition.

Note that a `FIELD GROUP` cannot be an alternate within a `CHOICE OF`.

A `nullable` qualifier can be used to indicate that a TLV Null can be encoded in place of the `CHOICE OF`. This is exactly the same as if NULL had been listed as one of the alternates.

Alternate Names and Tags

Alternates can be assigned textual names to distinguish them from one another. Each such name must be unique within the particular `CHOICE OF` definition. Alternate names do not affect the encoding of the resultant TLV. Rather, alternate names serve as user documentation, or as input to code generation tools.

Named `CHOICE OF` alternates may include at `tag` qualifier assigning a particular tag value to the alternate. When qualified in this way, the given tag value serves as a default tag for the alternate whenever the `CHOICE OF` appears in a context that doesn't otherwise specify a tag. The tags assigned within a `CHOICE OF` *do not* need to be unique, although see the discussion of Ambiguous Alternates below.

Both profile-specific and context-specific tags are allowed on the alternates of a `CHOICE OF` definition.

Nested CHOICE OFs and CHOICE OF Merging

It is legal for an alternate within a `CHOICE OF` to be another `CHOICE OF` definition, or a type reference to such. In this case, the effect is exactly as if the alternates of the inner `CHOICE OF` definition had been declared directly with the outer definition. This merging of `CHOICE OF` alternates occurs to any level of nesting, and can be used as a means of declaring `CHOICE OF`s that are supersets of other `CHOICE OF`s.

When alternates are merged, their names are preserved. In cases where the same name appears in nested `CHOICE OF` definitions, the name of the outer alternate is prepended to that of the inner alternate, separated by a dot, to form a unique name for the merged alternate.

Ambiguous Alternates

A `CHOICE OF` may contain multiple alternates having the same fundamental TLV type (e.g. two alternates that are both INTEGERs). If these alternates are also encoded using the same tag, their encoded forms are effectively indistinguishable from one another. Such alternates are referred to as *ambiguous alternates*.

Ambiguous alternates can occur due to the merging of nested `CHOICE OF` definitions (see above). They can also arise in cases where the tags associated with the alternates are overridden by a tag qualifier in an outer context; e.g. when a `STRUCTURE` incorporates a `CHOICE OF` field that has a specific tag qualifier assigned to the field.

Ambiguous alternates are legal in TLV schemes. However, care must be taken when introducing ambiguous alternates to ensure that a decoder can correctly interpret the resulting encoding. This can be achieved, for example, by signaling the appropriate interpretation via a data value (e.g. an enumerated integer) contained elsewhere in the encoding.

FIELD GROUP

Syntax

```
FIELD GROUP { field-group-members... }

field-group-member:
   identifier [ qualifiers... ]opt : type-or-ref      → field definition
   includes type-ref                                  → field group include
```

Allowed Qualifiers:

- **`FIELD GROUP`:** *none*
- *Field name identifier*: `tag`, `optional`

`FIELD GROUP` declares a collection of fields that may be included in a TLV structure. A `FIELD GROUP` is a pseudo-type that is never directly encoded in a TLV encoding. `FIELD GROUP`s are used with `includes` statements to define common patterns of fields such that they can be reused across different `STRUCTURE` definitions.

A `FIELD GROUP` definition contains a list of field definitions, each of which gives the type of the field, its tag, and an associated textual name. The field type can be either a fundamental type, a `CHOICE OF` pseudo-type, an `ANY` pseudo-type, or a reference to one of these types defined outside the `FIELD GROUP` definition.

A `FIELD GROUP` definition may also contain one or more `includes` statements. Each such statement identifies another `FIELD GROUP` whose fields are to be included within the referencing `FIELD GROUP`. Such nested inclusion can be specified to any depth.

The rules governing the names and tags associated with fields within a `FIELD GROUP` are the same as those defined for `STRUCTURE`s (see STRUCTURE).

Qualifiers

Qualifiers are annotations that provide additional information regarding the use or interpretation of a schema construct. Often qualifiers are used to place restrictions on the form or range of values that the construct can assume.

any-order / schema-order / tag-order

Syntax

```
STRUCTURE [ any-order ]
STRUCTURE [ schema-order ]
STRUCTURE [ tag-order ]
```

Allowed On: `STRUCTURE`

The `any-order`, `schema-order` and `tag-order` qualifiers can be used to specify a particular order for the encoding of fields within a TLV Structure. `tag-order` specifies that the fields of the structure must be encoded in increasing order of their numeric tag value. `schema-order` specifies that the fields of the structure must be encoded in the order listed within the `STRUCTURE` type. While `any-order` specifies that the fields of the structure can be encoded in any order.

Only a single ordering qualifier may be applied to a given `STRUCTURE` type.

For further information on the encoding order of TLV structures, see the Encoding Order section for `STRUCTURE` types.

extensible

Syntax

```
STRUCTURE [ extensible ]
```

Allowed On: STRUCTURE

The extensible qualifier is only allowed on STRUCTURE types, and declares that the structure can be extended by the inclusion of fields not listed in its definition. When a structure is extended in this way, any new fields that are included must use tags that are distinct from any defined field.

Example

```
user-information => STRUCTURE [ extensible ]
{
   user-id [1]             : UNSIGNED INTEGER,
   first-name [2]          : STRING,
   last-name [3]           : STRING,
   email-address [4]       : STRING,
}
```

id

Syntax

```
PROFILE [ id uint-value:uint-value ]          → 16-bit vendor id +
                                                16-bit profile number
PROFILE [ id name:uint-value ]                → vendor name +
                                                16-bit profile number
PROFILE [ id uint-value ]                     → 32-bit profile id

VENDOR [ id uint-value ]                      → 16-bit vendor id

MESSAGE [ id uint-value ]                     → 8-bit message number

STATUS CODE [ id uint-value ]                 → 16-bit status code
                                                number
```

Allowed On: PROFILE, VENDOR, MESSAGE, STATUS CODE

The `id` qualifier is used to specify an identifying number associated with a `PROFILE`, `VENDOR`, `MESSAGE` or `STATUS CODE` definition.

When applied to a `PROFILE` definition, the id value can take three forms: a single, 32-bit unsigned integer specifying the full profile id; two 16-bit unsigned integers (separated by a colon) specifying the Weave vendor id and profile number; or a vendor name plus a 16-bit profile number.

When applied to a `VENDOR` definition, the id value is a 16-bit unsigned integer specifying the Weave vendor id.

When applied to a `MESSAGE` definition, the id value is an 8-bit unsigned integer specifying the Weave message number.

When applied to a `STATUS CODE` definition, the id value is a 16-bit unsigned integer specifying the status code number.

length

Syntax

```
type [ length count ]                → exactly count
type [ length min..max ]             → between min and max
type [ length min.. ]                → at least min
```

Alias: `len`
Allowed On: `ARRAY`, `LIST`, `STRING`, `BYTE STRING`

The `length` qualifier can be used to constrain the number of elements in a collection type, such as an `ARRAY` or `LIST` , or the number of bytes in a `STRING` or `BYTE STRING` type.

nullable

Syntax

```
type [ nullable ]
```

Allowed On: `ARRAY`, `LIST`, `STRUCTURE`, `STRING`, `BYTE STRING`, `BOOLEAN`, `INTEGER`, `UNSIGNED INTEGER`, `FLOAT`

The `nullable` qualifier declares that a TLV null can be substituted for a value of the specified type at a particular point in an encoding. For example, in the following temperature sample

structure, a null value can be encoded for the cur-temp field (e.g. in the case the sensor was off-line at the sample time):

Example

```
temperature-sample => STRUCTURE
{
   timestamp [1] : UNSIGNED INTEGER,
   cur-temp [2] : FLOAT [ nullable ],
}
```

Applying a `nullable` qualifier to a type is exactly the same as defining a `CHOICE OF` type with alternates for the primary and NULL. For example, the temperature sample structure could also be defined as follows:

Example

```
temperature-sample => STRUCTURE
{
   timestamp [1] : UNSIGNED INTEGER,
   cur-temp [2] : CHOICE OF
      {
         FLOAT,
         NULL
      }
}
```

optional

Syntax

```
...
field-name [ optional ] : type-or-ref
...
```

Alias: `opt`
Allowed On: *Field definitions within* `STRUCTURE` *and* `FIELD GROUP` *types.*

The `optional` qualifier declares that a field within a `STRUCTURE` or `FIELD GROUP` is optional, and may be omitted by an encoder. The `optional` qualifier may only appear on the name portion of a field definition within either a `STRUCTURE` or `FIELD GROUP`.

Note that an optional field is distinct from one whose type has been declared `nullable`. In the former case the field may be omitted from the encoding altogether. In the latter case the

field must appear within the encoding, however its value may be replaced with Null. It is legal to declare a field that is both `optional` and `nullable`.

Example

```
user-information => STRUCTURE [ extensible ]
{
   user-id [1]                : UNSIGNED INTEGER,
   first-name [2]             : STRING,
   middle-name [3,optional]   : STRING,          → may be omitted
   last-name [4]              : STRING,
   email-address [5]          : STRING,
}
```

range

Syntax

```
integer-type [ range min..max ]                  → explicit constraint form
integer-type [ range 8bits ]                     → width constraint forms
integer-type [ range 16bits ]
integer-type [ range 32bits ]
integer-type [ range 64bits ]
```

Allowed On: `INTEGER`, `UNSIGNED INTEGER`, `FLOAT`

The `range` qualifier can be used to constrain the range of values for a numeric type such as `INTEGER` or `FLOAT`. Two forms are supported: *explicit constraints* and *width constraints*. Only one form can be applied to a given type.

An *explicit constraint* gives specific minimum and maximum values for the type. These can be any value that is legal for the underlying type.

A *width constraint* constrains the value to fit within a specific number of bits. Any of the width constraints (8, 16, 32 or 64 bits) can be applied to `INTEGER` and `UNSIGNED INTEGER` types; only 32 and 64 bit constraints can be applied to `FLOAT` types.

Note that a width constraint `range` qualifier does not obligate an encoder to always encode the specified number of bits. Per the TLV encoding rules, senders are always free to encode integer and floating point values in any encoding size, bigger or smaller, that will accommodate the value.

Example

```
system-status-event => STRUCTURE
{
   timestamp [1]            : UNSIGNED INTEGER [ range 32bits ]
   num-processes [1]        : UNSIGNED INTEGER [ range 16bits ],
   percent-busy [2]         : UNSIGNED INTEGER [ range 0..100 ],
}
```

tag

Syntax

```
identifier [ tag_opt tag-num ]                    → context-specific tag
identifier [ tag_opt profile-id:tag-num ]         → profile-specific tag
identifier [ tag_opt profile-name:tag-num ]       → profile-specific tag
identifier [ tag_opt *:tag-num ]                  → profile-specific tag (cur. profile)
identifier [ tag_opt anon ]                       → anonymous/no tag
```

Allowed On: type names, field names within a `STRUCTURE` or `FIELD GROUP`, item names within a `LIST`, alternate names within a `CHOICE OF`

The `tag` qualifier specifies a numeric tag value to be used when encoding a particular value. In Weave TLV, tags act as labels that identify the semantic meaning of the value to which they are attached. In Weave TLV schemas, `tag` qualifiers are applied to an identifier that names a type or an element of a type. The given value serves as a numeric alias for the textual name, which is used instead of the name when encoding the value.

Weave TLV supports two forms of tags: *profile-specific tags* and *context-specific tags*. Profile-specific tags contain a profile id and are globally unique, while context-specific tags are only unique within a particular TLV container.

As a special case, the keyword `anon` can be used to signal a value that should be encoded without a tag (or, more specifically, with the ‘anonymous’ tag).

For brevity, the `tag` keyword can be omitted when specifying a tag qualifier.

Context-specific Tags

A context-specific tag is a small integer value that identifies a TLV element within the context of a containing `STRUCTURE` or `LIST`. Values for context-specific tags are limited to the range 0 to 255 (8 bits), and can be given in either decimal or hexadecimal format.

Profile-specific Tags

A profile-specific tag is a colon-separated tuple containing a profile id and a tag number, each of which is a 32-bit unsigned integer. A profile-specific tag serves as a globally unique identifier for an encoded value that is unambiguous in all contexts.

The profile id portion of a profile-specific tag consists of the globally unique identifier for the Weave profile in which the tag is defined. The tag number, in turn, is a distinct integer value for the tag assigned within the context of the profile.

Profile id and tag number values can be given in either decimal or hexadecimal format. Profile ids can also be specified indirectly, by giving the name of a `PROFILE` definition located elsewhere in the schema. Finally, an asterisk (`*`) can be used as a shorthand to refer to the id of the `PROFILE` definition in which the `tag` qualifier appears. This profile is referred to as the *current profile*.

Explicit Tags

A `tag` qualifier that appears on a field within a `STRUCTURE` or `FIELD GROUP`, or on an item within a `LIST`, specifies the exact tag to be used when encoding the associated field/item. Such a tag is called an *explicit tag*, and may be either a context-specific or profile-specific tag.

If a field or item lacks a `tag` qualifier, then the encoding will use a default tag associated with the underlying field type, if such a tag has been specified. See the description of Default Tags in the next section.

Default Tags

A `tag` qualifier that appears on a type definition, or on an alternate within a `CHOICE OF`, serves as a default tag. A *default tag* is used to encode a value when an explicit tag has not been given in the schema.

For example, a field within a `STRUCTURE` that refers to a type with a default tag will use that tag if no `tag` qualifier has been specified on the field itself. Similarly, `tag` qualifiers that appear on the alternates of a `CHOICE OF` serve as default tags to be used when no other tag has been specified.

Both context-specific and profile-specific tags can be used as default tags.

Example

```
security => PROFILE [ id 4 ]
{

  ec-pub-key [4:1] => BYTE STRING          → default profile-specific tag using
                                             a numeric profile id

  ec-priv-key [security:2] => STRUCTURE    → default profile-specific tag using
  {                                          a name
    priv-key [1]     : BYTE STRING,        → explicit context-specific tag

    pub-key [2,opt] : ec-pub-key           → explicit tag overrides
                                             default tag on ec-pub-key

    curve            : CHOICE OF           → tag depends on choice of
    {                                        id or name
      id [3]         : UNSIGNED INTEGER,   → default tag if id chosen
      name [4]       : STRING              → default tag if name chosen
    }
  }

  ecdsa-sig [*:3] => STRUCTURE             → shorthand for security:3
  {
    r [1]            : BYTE STRING,
    s [2]            : BYTE STRING
  }

} // end security PROFILE
```

Documentation and Comments

TLV schemas can include inline annotations that support the automatic generation of reference documentation and the production of documented code. TLV schemas follow the Javadoc style of annotation wherein documentation is wrapped in the special multi-line comment markers `/**` and `*/`.

Example

```
/** Temperature sample structure */
temperature-sample => STRUCTURE
{
  timestamp [1] : UNSIGNED INTEGER,
  temperature [2] : FLOAT,
}
```

In certain cases, documentation can also be placed after a construct, using the `/**<` and `*/`.

Example

```
/** Temperature sample structure */
temperature-sample => STRUCTURE
{
  timestamp [1] : UNSIGNED INTEGER, /**< Unix timestamp */
  temperature [2] : FLOAT,          /**< Temperature in degrees C */
}
```

Postfix annotations are allowed on `STRUCTURE` and `FIELD GROUP` members, `ARRAY` and `LIST` items, `CHOICE OF` alternates and `INTEGER` enumerated values.

Non-documentation comments follow the standard C++ commenting style.

Example

```
user-information => STRUCTURE [ extensible ]
{
   user-id [1]        : UNSIGNED INTEGER, // 0 = Unknown user id
   first-name [2]     : STRING,
   last-name [3]      : STRING,
   email-address [4] : STRING,

   /* TODO: additional fields to be added later */
}
```

Revision History

Revision	Date	Description
0.9	2020/04/21	Complete specification, minus some introductory texts.
1.0	2020/05/03	Initial release version.