CommsDSL Specification v3.1.2

Table of Contents

Introduction . . . 3
- Specification Version . . . 3
- Schema Definition . . . 4
- Multiple Files . . . 7
- Namespaces . . . 8
- Platforms . . . 10
- References to Elements . . . 10
- Properties . . . 11
- Numeric Values . . . 12
- Boolean Values . . . 13
- Names . . . 13
- Protocol Versioning . . . 14

Schema . . . 15
- Protocol Name . . . 15
- Endian . . . 15
- Description . . . 15
- Protocol (Schema) Version . . . 16
- DSL Version . . . 16
- Allowing Non-Unique Message IDs . . . 17

Fields . . . 17
- Common Properties of Fields . . . 19
- <enum> Field . . . 24
- <int> Field . . . 30
- <set> Field . . . 39
- <bitfield> Field . . . 44
- <bundle> Field . . . 46
- <string> Field . . . 48
- <data> Field . . . 50
- <list> Field . . . 52
- <float> Field . . . 58
- <ref> Field . . . 64
- <optional> Field . . . 66
- <variant> Field . . . 70
- Referencing Values of Other Fields . . . 75

Messages . . . 83
- Message Name . . . 83

Numeric ID . . . 83
Description . . . 84
Display Name . . . 84
Fields . . . 85
Ordering . . . 87
Versioning . . . 87
Platforms . . . 88
Sender . . . 89
Customization . . . 90
Alias Names to Member Fields . . . 90
Interfaces . . . 90
Interface Name . . . 91
Description . . . 91
More About Fields . . . 91
Alias Names to Fields . . . 92
Aliases . . . 93
Description . . . 95
More on Aliases in <message>-es . . . 95
More on Aliases in <interface>-es . . . 96
More on Aliases in <bundle>-es . . . 96
Frames . . . 97
Name . . . 97
Description . . . 98
Layers . . . 98
Common Properties of Layers . . . 99
<payload> Layer . . . 101
<id> Layer . . . 102
<size> Layer . . . 102
<sync> Layer . . . 104
<checksum> Layer . . . 105
<value> Layer . . . 108
<custom> Layer . . . 112
Protocol Versioning Summary . . . 114
Version of the Schema . . . 114
Version in the Interface . . . 114
Version in the Frame . . . 114
Version of the Fields and Messages . . . 116
Version Dependency of the Code . . . 116
Compatibility Recommendation . . . 116
Appendix . . . 117
Properties of <schema> . . . 117

Common Properties of Fields . . . 117
Properties of <enum> Field . . . 119
Properties of <int> Field . . . 120
Properties of <set> Field . . . 123
Properties of <bitfield> Field . . . 124
Properties of <bundle> Field . . . 124
Properties of <string> Field . . . 125
Properties of <data> Field . . . 125
Properties of <list> Field . . . 126
Properties of <float> Field . . . 127
Properties of <ref> Field . . . 128
Properties of <optional> Field . . . 129
Properties of <variant> Field . . . 130
Units . . . 130
Properties of <message> . . . 132
Properties of <interface> . . . 134
Properties of <alias> . . . 134
Properties of <frame> . . . 135
Common Properties of Frame Layers . . . 135
Properties of <checksum> Frame Layer . . . 136
Properties of <value> Frame Layer . . . 136
Properties of <custom> Frame Layer . . . 137

Introduction

This document contains specification of **D**omain **S**pecific **L**anguage (DSL), called **CommsDSL**, for CommsChampion Ecosystem, used to define custom binary protocols. The defined schema files are intended to be parsed and used by commsdsl library and code generation application(s).

The PDF can be downloaded from release artifacts of from CommsDSL-Specification project. The online HTML documentation is hosted on github pages.

Specification Version

This document is versioned using Semantic Versioning.

The first (**MAJOR**) number in the version will describe the version of **DSL** itself, the second (**MINOR**) number will indicate **small** additions (such as adding new property for one of the

elements) to the specification which do not break any backward compatibility, and the third (**PATCH**) number (if exists) will indicate various language fixes and/or formatting changes of this specification document.

Schema Definition

The **CommsDSL** schema files use XML to define all the messages, their fields, and framing.

Every schema definition file must contain a valid XML with an encoding information header as well as **single** root node called <**schema**>:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    ...
</schema>
```

The schema node may define its properties (described in detail in Schema chapter).

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" version="2">
    ...
</schema>
```

Common Fields

It can also contain definition of various common fields that can be referenced by multiple messages. Such fields are defined as children of <**fields**> node.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeField> type="uint8" />
        ...
    </fields>
</schema>
```

There can be multiple <**fields**> elements in the same schema definition file. The fields are described in detail in Fields chapter.

Messages

The definition of a single message is done using <**message**> node (described in detail in Messages chapter).

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="SomeMessage" id="1">
        ...
    </message>

    <message name="SomeOtherMessage" id="2">
        ...
    </message>
</schema>
```

Multiple messages can (but don't have to) be bundled together as children of <**messages**> node.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <messages>
        <message name="SomeMessage" id="1">
            ...
        </message>

        <message name="SomeOtherMessage" id="2">
            ...
        </message>
    </messages>
</schema>
```

Framing

Transport framing is defined using <**frame**> node (described in detail in Frames chapter).

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="SomeFrame">
        <size ... />
        <id ... />
        <payload ... />
    </frame>
</schema>
```

Multiple frames can (but don't have to) be bundled together as children of <**frames**> node.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frames>
        <frame name="SomeFrame">
            ...
        </frame>

        <frame name="SomeOtherFrame">
            ...
        </frame>
    </frames>
</schema>
```

Interface

There are protocols that put some information, common to all the messages, such as protocol version and/or extra flags, into the framing information instead of message payload. This information needs to be accessible when message payload is being read or message object is being handled by the application. The COMMS Library handles these cases by having a common interface class for all the messages, which contains this extra information. In order to support such cases, the **CommsDSL** introduces optional node <**interface**> (described in detail in Interfaces chapter) for description of such common interfaces.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <interface name="CommonInterface">
        <int name="version" type="uint16" semanticType="version" />
    </interface>
</schema>
```

Multiple interfaces can (but don't have to) be bundled together as children of <**interfaces**> node.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <interfaces>
        <interface name="CommonInterface">
            ...
        </interface>

        <interface name="SomeOtherInterface">
            ...
        </interface>
    </interfaces>
</schema>
```

All the nodes described above are allowed to appear in any order.

Multiple Files

For big protocols it is possible and even recommended to split schema definition into multiple files. The code generator **must** accept a list of schema files to process and **must** process them in requested order.

Every subsequently processed schema file must **NOT** change any properties specified by the first processed schema file. It is allowed to omit any properties that have already been defined. For example:

First processed file:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" version="2">
    ...
</schema>
```

Second processed file:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    ...
</schema>
```

It must be accepted by the code generator because it does **NOT** change previously defined **name**, **endian**, and **version**. The second file doesn't mention **version** at all.

However, the following third file must cause an error due to changing the **endian** value:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="little">
    ...
</schema>
```

Also note, that all the properties have some default value and cannot be defined in second or later processed file while been omitted in the first one.

For example, the first file doesn't specify version number (which defaults to **0**)

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    ...
</schema>
```

The following second file must cause an error due to an attempt to override **version** property with different value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" version="2">
    ...
</schema>
```

Namespaces

In addition to splitting into multiple files, **CommsDSL** provides namespaces to help in definition of big protocols. It is possible to define fields, messages, interfaces, and frames in a separate namespace. The code generator must use this information to define relevant classes in a separate namespace(s) (if such feature is provided by the language) or introduce relevant prefixes into the names to avoid name clashes.

The namespace is defined using **<ns>** node with single **name** property. It can contain all the mentioned previously nodes.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <ns name="myns">
        <fields>
            ...
        </fields>

        <message ...>
        </message>

        <messages>
            <message ... />
            <message ... />
        </messages>

        <interface ...>
        </interface>

        <interfaces>
            <interface ... />
            <interface ... />
        </interfaces>

        <frame ...>
        </frame>

        <frames>
            <frame ... />
            <frame ... />
        </frames>
    </ns>
</schema>
```

The namespace (<**ns**>) can also contain other namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <ns name="myns">
        <ns name="subns1">
            <fields>
                ...
            </fields>
        </ns>

        <ns name="subns2">
            <message ... />
            <message ... />
        </ns>
    </ns>
</schema>
```

Platforms

The same protocol may be used by multiple **platforms** with a couple of platform specific messages. The **CommsDSL** allows listing of available platforms using optional <**platform**> node. Every message definition may specify a list of supported platforms. A code generator may use this information and generate some platform specific code.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <platform name="Plat1" />
    <platform name="Plat2" />
    <platform name="Plat3" />
</schema>
```

Multiple platforms can (but don't have to) be bundled together as children of <**platforms**> node.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <platforms>
        <platform name="Plat1" />
        <platform name="Plat2" />
        <platform name="Plat3" />
    </platforms>
</schema>
```

References to Elements

The **CommsDSL** allows references to fields or other definitions in order not to duplicate information and avoid various copy/paste errors. The referenced element **must** be defined (if in the

same file) or processed (if in different schema file) **before** the definition of the referencing element.

For example, message defines its payload as a reference (alias) to the globally defined field. This field is defined before the message definition. The opposite order **must** cause an error. It allows easy avoidance of circular references.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeField" type="uint8" />
    </fields>

    <message name="SomeMessage" id="1">
        <ref field="SomeField" />
    </message>
</schema>
```

When referencing a field or a value defined in a namespace (any namespace, not just different one), the former must be prefixed with a namespace(s) name(s) separated by a **.** (dot).

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <ns name="myns">
        <fields>
            <int name="SomeField" type="uint8" />
        </fields>

        <ns name="subns">
            <fields>
                <int name="SomeOtherField" type="uint16" />
            </fields>
        </ns>

        <message name="SomeMessage" id="1">
            <ref field="myns.SomeField" />
            <ref field="myns.subns.SomeOtherField" />
        </message>
    </ns>
</schema>
```

Properties

Almost every element in **CommsDSL** has one or more properties, such as **name**. Any property can be defined using multiple ways. In can be useful when an element has too many properties to specify in a single line for a convenient reading. **Any** of the described below supported ways of defining a single property can be used for **any** element in the schema.

The property can be defined as an XML attribute.

```
<int name="SomeField" type="uint8" />
```

Or as child node with **value** attribute:

```
<int>
    <name value="SomeField" />
    <type value="uint8" />
</int>
```

Property value can also be defined as a text of the child XML element.

```
<int>
    <name>SomeField</name>
    <type>uint8</type>
</int>
```

It is allowed to mix ways of defining properties for a single element

```
<int name="SomeField">
    <type value="uint8" />
    <endian>big</endian>
</int>
```

Many properties must be defined only once for a specific element. In this case, repetition of it is prohibited. The definition below must cause an error (even if provided **type** value is not changed).

```
<int name="SomeField" type="uint8" >
    <type value="uint8" />
</int>
```

NOTE, that properties can be defined in **any** order.

Numeric Values

Any integral numeric value in the schema may be defined as decimal value or hexadecimal with "0x" prefix. For example, numeric IDs of the messages below are specified using decimal (for first) and hexadecimal (for second).

```
<message name="Message1" id="123">
    ...
</message>

<message name="Message2" id="0x1a">
    ...
</message>
```

Boolean Values

There are properties that require boolean value. The **CommsDSL** supports case **insensitive** "true" and "false" strings, as well as "1" and "0" numeric values.

```
<int name="SomeField" ... removed="True" />
<int name="SomeOtherField" ... pseudo="0" />
```

Names

Almost every element has a required **name** property. Provided value will be used to generate appropriate classes and/or relevant access functions. As the result, the chosen names must be only alphanumeric and `_' (underscore) characters, but also mustn't start with a number. The provided value is case sensitive. However, the code generator is allowed to change the case of to first letter of the provided value. It is up to the code generator to choose whether to use **camelCase** or **PascalCase** when generating appropriate classes and/or access functions.

As the result, the code generator may report an error for the following definition of fields, which use different names, but differ only in the case of the first letter.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="someField" type="uint8" />
        <int name="SomeField" type="uint16" />
    </fields>
</schema>
```

The names of the elements must be unique in their scope. It is allowed to use the same name in different namespaces or different messages.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        <int name="F1"> type="uint8" />
        ...
    </message>

    <message name="Msg2" id="2">
        <int name="F1"> type="int32" />
        ...
    </message>
</schema>
```

Protocol Versioning

The **CommsDSL** provides a way to specify version of the binary protocol by using **version** property of the schema element.

Other elements, such as fields or messages allow specification of version they were introduced by using **sinceVersion** property. It is also possible to provide an information about version since which the element has been deprecated using **deprecated** property. Usage of **deprecated** property is just an indication for developers that the element should not be used any more. The code generator may introduce this information as a comment in the generated code. However, it does **NOT** remove a deprecated field from being serialized to preserve backward compatibility of the protocol. If the protocol definition does require removal of the deprecated field from being serialized, the **deprecated** property must be supplemented with **removed** property.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" version="5" >
    <message name="SomeMessage" id="1">
        <int name="F1" type="uint16" />
        <int name="F2" type="uint8" sinceVersion="2" />
        <int name="F3" type="int32" sinceVersion="3" deprecated="4" removed="true" />
    </message>
</schema>
```

In the example above the field **F2** was introduced in version 2. The field **F3** was introduced in version 3, but deprecated and removed in version 4.

All these version numbers in the schema definition allow generation of proper version checks and correct code for protocols that communicate their version in their framing or selected messages. Please refer to Protocol Versioning Summary chapter for more details on the subject.

For all other protocols that don't report their version and/or don't care about backward compatibility, the version information in the schema just serves as documentation. The code

generator must ignore the version information when generating code for such protocols. The code generator may also allow generation of the code for a specific version and take provided version information on determining whether specific field exists for a particular version.

Schema

Schema definition may contain various global (protocol-wide) properties.

Protocol Name

The protocol name is defined using **name** property. It may contain any alphanumeric character, but mustn't start with a number.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol">
    ...
</schema>
```

The **name** property is a **required** one. The code generator must report an error in case first processed schema file doesn't define one.

The code generator is expected to use the specified name as main namespace for the protocol definition, unless new name is provided via command line parameters.

Endian

Default endian for the protocol can be defined using **endian** property. Supported values are either **big** or **little** (case insensitive). Defaults to **little**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="Big">
    ...
</schema>
```

The **endian** property of any subsequently defined field will default to the specified value, but can be overridden using its own **endian** property.

Description

It is possible to provide a human readable description of the protocol definition just for documentation purposes. The description is provided using **description** property of the <**schema**> node. Just like any property the description can be provided using one of the accepted ways. In case of long multiline description it is recommended to define it as a text child element.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol">
    <description>
        Some
        multiline
        description
    </description>
    ...
</schema>
```

Protocol (Schema) Version

As was mentioned in Protocol Versioning section, **CommsDSL** supports (but doesn't enforce) versioning of the schema / protocol. In order to specify the version use **version** property with unsigned integral value. Defaults to **0**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" version="5">
    ...
</schema>
```

In case the protocol definition uses semantic versioning with major / minor numbers, it is recommended to combine multiple numbers into one mentally using "shift" operation(s). For example version **1.5** can be defined and used throughout the schema as **0x105**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" version="0x105">
    ...
</schema>
```

DSL Version

As this specification evolves over time it can introduce new properties or other elements. It is possible to specify the version of the **DSL** as the schema's property. If code generator expects earlier version of the schema it should report an error (or at least a warning).

The DSL version is specified using **dslVersion** property with unsigned integral value. Defaults to **0**, which means any version of code generator will try to parse the schema and will report error / warning in case it encounters unrecognized property or other construct.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" dslVersion="2">
    ...
</schema>
```

Allowing Non-Unique Message IDs

By default every defined message must have unique numeric message ID. If this is not the case, the code generator must report an error in case message definition with repeating ID number is encountered. It is done as protection against various copy/paste or typo errors.

However, there are protocols that may define various forms of the same message, which are differentiated by a serialization length or value of some particular field inside the message. It can be convenient to define such variants as separate classes. **CommsDSL** allows doing so by setting **nonUniqueMsgIdAllowed** property of the schema to **true**. In this case, code generator must allow definition of different messages with the same numeric ID.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" nonUniqueMsgIdAllowed="true">
    <message name="SomeMessageForm1" id="1">
        ...
    </message>

    <message name="SomeMessageForm2" id="1">
        ...
    </message>
</schema>
```

Use properties table for future references.

Fields

Any **field** can be defined as independent element inside the <**fields**> child of the <schema> or a <ns> .

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeField" type="uint8" />
        ...
    </fields>
</schema>
```

It can also be defined as a member of a message.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="SomeMessage" id="1">
        <int name="SomeField" type="uint8" />
        ...
    </message>
</schema>
```

Field that is defined as a child of <**fields**> node of the <schema> or <ns> can be referenced by other fields to avoid duplication of the same definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <ns name="ns1">
        <int name="SomeField" type="uint8" />
        <ref name="AliasToField" field="ns1.SomeField" />
    </ns>
    <message name="SomeMessage" id="1">
        <ref name="Mem1" field="ns1.SomeField"" />
        ...
    </message>
</schema>
```

The available fields are described in details in the sections to follow. They are:

- <enum> - Enumeration field.
- <int> - Integral value field.
- <set> - Bitset (bitmask) field.
- <bitfield> - Bitfield field.
- <bundle> - Bundle field.
- <string> - String field.
- <data> - Raw data field.
- <list> - List of other fields.
- <float> - Floating point value field.
- <ref> - Reference to (alias of) other field.
- <optional> - Optional field.
- <variant> - Variant field.

All this fields have common as well as their own specific set of properties.

Common Properties of Fields

Every field is different, and defines its own properties and/or other aspects. However, there are common properties, that are applicable to every field. They are summarized below.

Name

Every field must define it's name, which is expected to be used by a code generator when defining a relevant class. The name is defined using **name** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeField" ... />
    </fields>
</schema>
```

Description

It is possible to provide a description of the field about what it is and how it is expected to be used. This description is only for documentation purposes and may find it's way into the generated code as a comment for the generated class. The property is **description**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeField" type="uint8">
            <description>
                Some long
                multiline
                description
            </description>
        </int>
    </fields>
</schema>
```

Reusing Other Fields

Sometimes two different fields are very similar, but differ in one particular aspect. **CommsDSL** allows copying all the properties from previously defined field (using **reuse** property) and change some of them after the copy. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint8" defaultValue="3" />
        <bitfield name="SomeBitfield">
            <int name="Member1" reuse="SomeIntField" bitLength="3" />
            <int name="Member2" type="uint8" defaultValue="10" bitLength="5" />
        </bitfield>
    </fields>
</schema>
```

In the example above member of the bitfield **Member1** copies all the properties from **SomeIntField** field, then overrides its **name** and adds **bitLength** one to specify its length in bits.

The **reuse** is allowed only from the field of the same **kind**. For example, it is **NOT** allowed for <enum> field to reuse definition of <int> , only other <enum> .

Forcing Generation

By default the code generator is expected not to generate code for fields that are not referenced by any other field to reduce amount of generated code. However, there may be cases when a field releated definitions are expected to find their way into the generated code even the field itself is not referenced anywhere. To help with such forcing the **forceGen** property has been introduced with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" forceGen="true" ... />
    </fields>
</schema>
```

Display Properties

CommsDSL supports generation of not only field's serialization and value access functionality, but also of various GUI protocol visualization, debugging and analysis tools. There are some allowed properties, that indicate how the field is expected to be displayed by such tools.

The **displayName** property is there to specify proper string of the field's name, with spaces, dots and other characters that are not allowed to exist in the name.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" displayName="Some Int Field" ... />
    </fields>
</schema>
```

If **displayName** is not specified, the code generator must use value of property **name** instead. In order to force empty **displayName**, use "_" (underscore) value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" displayName="_" ... />
    </fields>
</schema>
```

Sometimes the values of some fields may be controlled by other fields. In this case, it could be wise to disable manual update of such fields. To enable/disable such behavior use **displayReadOnly** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" displayReadOnly="true" ... />
    </fields>
</schema>
```

Also sometimes it can be desirable to completely hide some field from being displayed in the protocol analysis GUI application. In this case use **displayHidden** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" displayHidden="true" ... />
    </fields>
</schema>
```

Versioning

CommsDSL allows providing an information in what version the field was added to a particular message, as well as in what version it was deprecated, and whether it was removed from being serialized after deprecation.

To specify the version in which field was introduced, use **sinceVersion** property. To specify the

version in which the field was deprecated, use **deprecated** property. To specify whether the field was removed after being deprecated use **removed** property in addition to **deprecated**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" version="5" >
    <message name="SomeMessage" id="1">
        <int name="F1" type="uint16" />
        <int name="F2" type="int8" deprecated="5" />
        <int name="F3" type="uint8" sinceVersion="2" />
        <int name="F4" type="int32" sinceVersion="3" deprecated="4" removed="true" />
    </message>
</schema>
```

In the example above:

- **F1** was introduced in version **0** and hasn't been deprecated yet.
- **F2** was also introduced in version **0**, deprecated in version **5**, but **not** removed from being serialized.
- **F3** was introduced in version **2** and hasn't been deprecated yet.
- **F4** was introduced in version **3**, deprecated in removed in version **4**.

NOTE, that all the specified versions mustn't be greater that the version of the schema. Also value of **sinceVersion** must be **less** than value of **deprecated**.

The version information on the field in global **<fields>** area or inside some namespace does **NOT** make sense and should be ignored by the code generator. It is allowed when field is a member of a <message> or a <bundle> field.

Failing Read of the Field on Invalid Value

Some fields may specify what values are considered to be valid, and there may be a need to fail the **read** operation in case the received value is invalid.

To achieve this **failOnInvalid** property with boolean value can be used. There are two main scenarios that may require usage of this property. One is the protocol being implemented requires such behavior in its specification. The second is when there are multiple forms of the same message which are differentiated by the value of some specific field in its payload. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" nonUniqueMsgIdAllowed="true" >
    <message name="Msg1Kind0" id="1" order="0">
        <int name="Kind" type="uint8" validValue="0" failOnInvalid="true" />
        ...
    </message>

    <message name="Msg1Kind1" id="1" order="1">
        <int name="Kind" type="uint8" defaultValue="1" validValue="1"
failOnInvalid="true" />
        ...
    </message>

    <message name="Msg1Kind2" id="1" order="2">
        <int name="Kind" type="uint8" defaultValue="2" validValue="2"
failOnInvalid="true" />
        ...
    </message>
</schema>
```

The example above defined 3 variants of the message with numeric ID equals to **1**. When new message with this ID comes in, the framing code is expected to try reading all of the variants and choose one, on which **read** operation doesn't fail. The **order** property of the message specifies in what order the messages with the same ID must be read. It described in more detail in Messages chapter.

Pseudo Fields

Sometimes there may be a need to have "psuedo" fields, which are implemented using proper field abstration, and are handled as any other field, but not actually getting serialized when written (or deserialized when read). It can be achieved using **pseudo** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" version="5" >
    <message name="SomeMessage" id="1">
        <int name="SomePseudoField" type="uint16" defaultValue="0xabcd" pseudo="true"
/>
        <int name="SomeRealField" type="int8">
        ...
    </message>
</schema>
```

Customizable Fields

The code generator is expected to allow some level of compile time customization of the generated code, such as choosing different data structures and/or adding/replacing some runtime logic. The code generator is also expected to provide command line options to choose required level of

customization. Sometimes it may be required to allow generated field abstraction to be customizable regardless of the customization level requested from the code generator. **CommsDSL** provides **customizable** property with boolean value to force any field being customizable at compile time.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="SomeField" customizable="true" />
    </fields>
</schema>
```

Semantic Type

Sometimes code generator may generate a bit different (or better) code for fields that are used for some particular purpose. To specify such purpose use **semanticType** property.

Available semantic types are:

- **messageId** - Used to specify what type/field is used for holding numeric message ID. Applicable to <enum> fields.
- **version** - Used to specify that the field is used to hold protocol version. Applicable to <int> field (or <ref> referening an <int>).
- **length** - Used to specify that the field holds total serialization length of the subsequent fields. Applicable to <int> field (or <ref> referening an <int>). The **length** semantic type makes sense only for a member of <bundle> field and should be ignored in all other cases. The **length** semantic type was introduced in **v2.0** of **CommsDSL** specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="MsgId" type="uint8" semanticType="messageId" >
            <validValue name="Msg1" val="0x01" />
            <validValue name="Msg2" val="0x02" />
            <validValue name="Msg3" val="0x03" />
            ...
        </enum>
    </fields>
</schema>
```

Use properties table for future references.

<enum> Field

This field stores and abstracts away value of integral enumerated type, where every valid value has its name. The <**enum**> field has all the common properties as well as extra properties and elements

described below.

Underlying Type

Every <**enum**> field must provide its underlying storage type using **type** property. Available values are:

- **int8** - 1 byte signed integer.
- **uint8** - 1 byte unsigned integer.
- **int16** - 2 bytes signed integer.
- **uint16** - 2 bytes unsigned integer.
- **int32** - 4 bytes signed integer.
- **uint32** - 4 bytes unsigned integer.
- **int64** - 8 bytes signed integer.
- **uint64** - 8 bytes unsigned integer.
- **intvar** - up to 8 bytes variable length signed integer
- **uintvar** - up to 8 bytes variable length unsigned integer

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="SomeEnumField" type="uint8">
            ...
        </enum>
    </fields>
</schema>
```

The variable length types are encoded using **Base-128** form, such as LEB128 for **little** endian or similar for **big** endian.

Valid Values

All the valid values must be listed as <**validValue**> child of the <**enum**> XML element.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="SomeEnumField" type="uint8">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="Val3" val="0x1b" />
        </enum>
    </fields>
</schema>
```

Every <**validValue**> must define a valid name (using **name** property) as well as numeric value (using **val** property), that fits chosen underlying type. The <**validValue**>-es may be listed in any order, not necessarily sorted.

Every <**validValue**> has extra optional properties:

- **description** - Extra description and documentation on how to use the value.
- **displayName** - String specifying how to name the value in various analysis tools.
- **sinceVersion** - Version of the protocol when the value was introduced.
- **deprecated** - Version of the protocol when the value was deprecated.

All these extra properties are described in detail in Common Properties of Fields.

Default Value

The default value of the <**enum**> field when constructed can be specified using **defaultValue** property. If not specified, defaults to **0**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="SomeEnumField" type="uint8" defaultValue="5">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="Val3" val="0x1b" />
        </enum>
    </fields>
</schema>
```

The default value can also be specified using the name of one of the <**validValue**>-es:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="SomeEnumField" type="uint8" defaultValue="Val2">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="Val3" val="0x1b" />
        </enum>
    </fields>
</schema>
```

Endian

The default serialization endian of the protocol is specified in **endian** property of the schema. It is possible to override the default endian value with extra **endian** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <enum name="SomeEnumField" type="uint16" endian="little">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="Val3" val="0x1b" />
        </enum>
    </fields>
</schema>
```

Serialization Length

The underlying type dictates the serialization length of the **<enum>** field. However, there may be protocols that limit serialization length of the field to non-standard lengths, such as **3** bytes. In this case use **length** property to specify custom serialization length.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <enum name="SomeEnumField" type="uint32" length="3">
            <validValue name="Val1" val="0x0" />
            <validValue name="Val2" val="0x0a0b0c" />
            <validValue name="Val3" val="0xffffff" />
        </enum>
    </fields>
</schema>
```

IMPORTANT: When **length** property is used with variable length underlying type (**intvar** and **uintvar**), it means **maximum** allowed length.

Length in Bits

<enum> field can be a member of <bitfield> field. In this case the serialization length may be specified in bits using **bitLength** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <bitfield name="SomeBitfield">
            <enum name="SomeEnumMember" type="uint8" bitLength="3">
                <validValue name="Val1" val="0" />
                <validValue name="Val2" val="1" />
                <validValue name="Val3" val="2" />
            </enum>

            <enum name="SomeOtherEnumMember" type="uint8" bitLength="5">
                <validValue name="Val1" val="5" />
                <validValue name="Val2" val="12" />
                <validValue name="Val3" val="20" />
            </enum>
        </bitfield>
    </fields>
</schema>
```

Hex Assignment

The code generator is expected to generate appropriate **enum** types using **decimal** values assigned to enumeration names. However, some protocol specifications may list valid values using **hexadecimal** format. To make the reading of the generated code more convenient, use **hexAssign** property with boolean value to force code generator make the assignments using hexadecimal values.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <enum name="SomeEnumField" type="uint8" hexAssign="true">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="Val3" val="0x1b" />
        </enum>
    </fields>
</schema>
```

The generated enum type is expected to look something like this:

```
enum class SomeEnumFieldVal
{
    Val1 = 0x00,
    Val2 = 0x05,
    Val3 = 0x1b
};
```

instead of

```
enum class SomeEnumFieldVal
{
    Val1 = 0,
    Val2 = 5,
    Val3 = 27
};
```

Allow Non-Unique Values

By default, non-unqiue values are not allowed, the code generator must report an error if two different <**validValue**>-es use the same value of the **val** property. It is done as protection against copy-paste errors. However, **CommsDSL** allows usage of non-unique values in case **nonUniqueAllowed** property has been set to **true**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <enum name="SomeEnumField" type="uint8" nonUniqueAllowed="true">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="OtherNameForVal2" val="5" />
        </enum>
    </fields>
</schema>
```

Version Based Validity

The code generator is expected to generate functionality checking that <**enum**> field contains a valid value. By default any specified <**validValue**> is considered to be valid regardless of version it was introduced and/or deprecated. However, it is possible to force code generator to generate validity check code that takes into account reported version of the protocol by using **validCheckVersion** property, which is set to **true**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" version="5">
    <fields>
        <enum name="SomeEnumField" type="uint8" validCheckVersion="true">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="Val3" val="10" sinceVersion="2" />
            <validValue name="Val4" val="15" sinceVersion="3" deprecated="4"/>
        </enum>
    </fields>
</schema>
```

In the example above values **0** and **5** will always be considered valid. However value **10** will be considered valid only if reported protocol version is greater than or equal to **2**. The value **15** will be considered valid only for protocol version **3**.

Use properties table for future references.

<int> Field

This field stores and abstracts away value of integral type. The <**int**> field has all the common properties as well as extra properties and elements described below.

Underlying Type

Every <**int**> field must provide its underlying storage type using **type** property. Available values are:

- **int8** - 1 byte signed integer.
- **uint8** - 1 byte unsigned integer.
- **int16** - 2 bytes signed integer.
- **uint16** - 2 bytes unsigned integer.
- **int32** - 4 bytes signed integer.
- **uint32** - 4 bytes unsigned integer.
- **int64** - 8 bytes signed integer.
- **uint64** - 8 bytes unsigned integer.
- **intvar** - up to 8 bytes variable length signed integer
- **uintvar** - up to 8 bytes variable length unsigned integer

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntield" type="uint8" />
    </fields>
</schema>
```

The variable length types are encoded using **Base-128** form, such as LEB128 for **little** endian or similar for **big** endian.

Special Values

Some protocol may assign a special meaning for some values. For example, some field specifies configuration of some timer duration, when **0** value means infinite. Such values (if exist) must be listed as <**special**> child of the <**int**> XML element.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="Duration" type="uint8">
            <special name="Infinite" val="0" />
        </int>
    </fields>
</schema>
```

The code generator is expected to generate extra convenience functions that check whether field has special value as well as updating the stored value with special one.

Every <**special**> must define a valid name (using **name** property) as well as numeric value (using **val** property), that fits chosen underlying type. The <**special**>-s may be listed in any order, not necessarily sorted.

Every <**special**> has extra optional properties:

- **description** - Extra description and documentation on how to use the value.
- **sinceVersion** - Version of the protocol when the special name / meaning was introduced.
- **deprecated** - Version of the protocol when the special name / meaning was deprecated.
- **displayName** - Readable name to display for the special value in protocol debugging and visualization tools.

All these extra properties are described in detail in Common Properties of Fields.

By default, non-unqiue special values (different name for the same value) are not allowed, the code generator must report an error if two different <**special**>-es use the same value of the **val** property. It is done as protection against copy-paste errors. However, **CommsDSL** allows usage of non-unique values in case **nonUniqueSpecialsAllowed** property has been set to **true**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeInt" type="uint8" nonUniqueSpecialsAllowed="true">
            <special name="S1" val="0" />
            <special name="S2" val="0" />
        </int>
    </fields>
</schema>
```

Default Value

The default value of the <**int**> field when constructed can be specified using **defaultValue** property. If not specified, defaults to **0**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint8" defaultValue="5" />
    </fields>
</schema>
```

The default value can also be specified using the name of one of the <**special**>-s:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint8" defaultValue="Special2">
            <special name="Special1" val="0" />
            <special name="Special2" val="0xff" />
        </int>
    </fields>
</schema>
```

Endian

The default serialization endian of the protocol is specified in **endian** property of the schema. It is possible to override the default endian value with extra **endian** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <int name="SomeIntField" type="uint16" endian="little" />
    </fields>
</schema>
```

Serialization Length

The underlying type dictates the serialization length of the <**int**> field. However there may be protocols, that limit serialization length of the field to non-standard lengths, such as **3** bytes. In this case use **length** property to specify custom serialization length.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <int name="SomeIntField" type="uint32" length="3" />
    </fields>
</schema>
```

IMPORTANT: When **length** property is used with variable length underlying types (**intvar** and

uintvar), it means **maximum** allowed length.

Length in Bits

<int> field can be a member of <bitfield> field. In this case the serialization length may be specified in bits using **bitLength** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <bitfield name="SomeBitfield">
            <int name="SomeIntMember" type="uint8" bitLength="2" />
            <int name="SomeOtherIntMember" type="uint8" bitLength="6" />
        </bitfield>
    </fields>
</schema>
```

Serialization Offset

Some protocols may require adding/subtracting some value before serialization, and performing the opposite operation when the field is deserialized. Such operation can be forced using **serOffset** property with numeric value. The classic example would be defining a **year** field that is being serialized using 1 byte as offset from year 2000. Although it is possible to define such field as 1 byte integer

```
<int name="Year" type="uint8"  />
```

it is quite inconvenient to work with it in a client code. The client code needs to be aware what offset needs to be added to get the proper year value. It is much better to use **serOffset** property to manipulate value before and after serialization.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <int name="Year" type="int16" defaultValue="2000" serOffset="-2000" length="1"
/>
    </fields>
</schema>
```

NOTE, that value of **serOffset** property must fit into the underlying type defined using **type** property.

Sign Extension

When limiting serialization length using **length** property, the performed **read** operation is expected to sign extend read signed value. However, such default behavior may be incorrect for some cases,

especially when serialization offset is also used. There are protocols that disallow serialization of a negative value. Any signed integer must add predefined offset to make it non-negative first, and only then serialize. The deserialization procedure is the opposite, first deserialize the non-negative value, and then subtract predefined offset to get the real value.

For example, there is an integer field with expected valid values between `-8,000,000` and `+8,000,000`. This range fits into 3 bytes, which are used to serialize such field. Such field is serialized using the following math:

- Add 8,000,000 to the field's value to get non-negative number.
- Serialize the result using only 3 bytes.

In order to implement such example correctly there is a need to switch off the automatic sign extension when value is deserialized.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeField" type="int32" serOffset="8000000" length="3"
signExt="false" />
    </fields>
</schema>
```

NOTE, that **signExt** property is relevant only for signed types with non-default serialization length.

Scaling

Some protocols may not support serialization of floating point values, and use scaling instead. It is done by multiplying the original floating point value by some number, dropping the fraction part and serializing the value as integer. Upon reception, the integer value is divided by predefined number to get a proper floating point value.

For example, there is a distance measured in millimeters with precision of 4 digits after decimal point. The value is multiplied by 10,000 and serialized as **<int>** field. Such scenario is supported by **CommsDSL** via introduction of **scaling** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="Distance" type="uint32" scaling="1/10000" />
    </fields>
</schema>
```

NOTE, that format of **scaling** value is "**numerator** / **denominator**". The code generator is expected to define such field like any other **<int>**, but also provide functions that allow set / get of scaled floating point value.

It is possible to omit the **denominator** value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" type="int16" scaling="4" />
    </fields>
</schema>
```

In the example above it is equivalent to having **scaling="4/1"** defined.

Units

Protocols quite often specify what units are being transfered. The **CommsDSL** provides **units** property to specify this information. The code generator may use this information to generate a functionality that allows retrieval of proper value for requested units, while doing all the conversion math internally. Such behavior will allow developers, that use generated protocol code, to focus on their business logic without getting into details on how value was transfered and what units are used by default.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="Distance" type="uint32" units="mm" />
    </fields>
</schema>
```

For list of supported **units** values, refer to appended units table.

Quite often, **units** and **scaling** need to be used together. For example

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="Latitude" type="int32" units="deg" scaling="1/10000000" />
    </fields>
</schema>
```

The code generator may generate code that allows retrieval of proper (floating point) value of either **degrees** or **radians**, while all the scaling and conversion math is done automatically.

Valid Values

Many protocols specify ranges of values the field is allowed to have and how client code is expected to behave on reception of invalid values. The code generator is expected to generate code that checks whether field's value is valid. The **CommsDSL** provides multiple properties to help with such task.

One of such properties if **validRange**. The format of it's value is "[**min_value**, **max_value**]".

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint8" validRange="[0, 10]" />
    </fields>
</schema>
```

It is possible to have multiple valid ranges for the same field. However XML does NOT allow having multiple attributes with the same name. As the result it is required to put extra valid ranges as <**validRange**> children elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint8">
             <validRange value="[0, 10]" />
             <validRange value="[25, 40]" />
        </int>
    </fields>
</schema>
```

Another property is **validValue**, which adds single value (not range) to already defined valid ranges / values. Just like with **validRange**, multiple values need to be added as XML children elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint8" validRange="[0, 10]" validValue="15">
            <validValue value="40" />
        </int>
    </fields>
</schema>
```

There are also **validMin** and **validMax**, which specify single numeric value and are equivalent to having
`validRange="[provided_min_value, max_value_allowed_by_type]"` and
`validRange="[min_value_allowed_by_type, provided_max_value]"` respectively.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" type="int8" validMin="-20" />
        <int name="SomeOtherIntField" type="int8" validMax="100" />
    </fields>
</schema>
```

The specified valid ranges and values are allowed to intersect. The code generator may warn about such cases and/or unify them to limit number of **if** conditions in the generated code for better performance.

If none of the mentioned above validity related options has been used, the whole range of available values is considered to be valid.

All the validity related properties mentioned in this section (**validRange**, **validValue**, **validMin**, and **validMax**) may also add information about version they were introduced / deprecated in. Adding such information is possible only when the property is defined as XML child element.

```
<?xml version="1.0" encoding="UTF-8"? version="10">
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint8">
             <validRange value="[0, 10]" />
             <validValue value="25" sinceVersion="2" deprecated="5" />
             <validRange value="[55, 80]" sinceVersion="7" />
        </int>
    </fields>
</schema>
```

The **sinceVersion** and **deprecated** properties are described in detail as Common Properties of Fields.

Version Based Validity

The code generator is expected to generate functionality checking that **<int>** field contains a valid value. By default if the field's value is within any of the specified ranges / values, then the it is considered to be valid regardless of version the containing range was introduced and/or deprecated. However, it is possible to force code generator to generate validity check code that takes into account reported version of the protocol by using **validCheckVersion** property, which is set to **true**.

```
<?xml version="1.0" encoding="UTF-8"? version="10">
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint8" validCheckVersion="true">
             <validRange value="[0, 10]" />
             <validValue value="25" sinceVersion="2" deprecated="5" />
             <validRange value="[55, 80]" sinceVersion="7" />
        </int>
    </fields>
</schema>
```

Extra Display Properties

When scaling information is specified, it is possible to notify GUI analysis tools that value of <**int**> field should be displayed as scaled floating point number. To do so, use **displayDecimals** property with numeric value of how many digits need to be displayed after decimal point.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="Distance" type="uint32" scaling="1/10000" displayDecimals="4" />
    </fields>
</schema>
```

Also when serialization offset is provided, sometimes it may be desirable to display the value in the GUI analysis tool(s) with such offset.

For example, many protocols define some kind of remaining length field when defining a transport frame or other places. Sometimes the value of such field should also include its own length. However, it is much more convenient to work with it, when the retrieved value shows only **remaining** length of subsequent fields, without worrying whether the value needs to be reduced by the serialization length of holding field, and what exactly this length is. Such field can be defined like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="RemLength" type="uint16" serOffset="2" />
    </fields>
</schema>
```

In the example above, the field is expected to hold only **remaining** length, **excluding** the length of itself, but adding it when value is serialized.

However, when such field is displayed in GUI analysis tool(s), it is desirable to display the value with serialization offset as well. It can be achieved using **displayOffset** property with numeric value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="RemLength" type="uint16" serOffset="2" displayOffset="2"/>
    </fields>
</schema>
```

When <**special**> values are specified the protocol analysis tools are expected to display them next to actual numeric value of the field. The **displaySpecials** property with boolean value is there to control this default behavior.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeInt" type="uint8" displaySpecials="false">
            <special name="S1" val="0" />
            <special name="S2" val="2" />
        </int>
    </fields>
</schema>
```

Use properties table for future references.

<set> Field

This field stores and abstracts away value of bitset (bitmask) type. The **<set>** field has all the common properties as well as extra properties and elements described below.

Underlying Type

The underlying type of the **<set>** field can be provided its underlying storage type using **type** property. Available values are:

- **uint8** - 1 byte unsigned integer.
- **uint16** - 2 bytes unsigned integer.
- **uint32** - 4 bytes unsigned integer.
- **uint64** - 8 bytes unsigned integer.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetield" type="uint8">
            ...
        </set>
    </fields>
</schema>
```

NOTE that the available types are all fixed length unsigned ones.

Serialization Length

The underlying type specification may be omitted if serialization length (in number of bytes) is specified using **length** property. In this case underlying type is automatically selected based on the provided serialization length.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetield" length="1">
            ...
        </set>
    </fields>
</schema>
```

Length in Bits

<set> field can be a member of <bitfield> field. In this case the serialization length may be specified in bits using **bitLength** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <bitfield name="SomeBitfield">
            <int name="SomeIntMember" type="uint8" bitLength="4" />
            <set name="SomeSetMember" bitLength="4" >
                ...
            </set>
        </bitfield>
    </fields>
</schema>
```

NOTE that the underlying type information can be omitted when **bitLength** property is in use, just like with **length**.

Bits

The **<set>** field may list its bits as **<bit>** XML child elements. Every such element must specify its name using **name** property as well as its index using **idx** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetField" length="1">
            <bit name="SomeBitName" idx="0" />
            <bit name="SomeOtherBitName" idx="1" />
        </set>
    </fields>
</schema>
```

The bit indexing starts from **least** significant bit, and mustn't exceed number of bits allowed by the underlying type. The **<bit>**-s may be listed in any order, not necessarily sorted.

The code generator is expected to generate convenience functions (or other means) to set / get the value of every listed bit.

Every <**bit**> element may also define extra properties listed below for better readability:

- **description** - Extra description and documentation on the bit.
- **displayName** - String specifying how to name the bit in various analysis tools.

These properties are described in detail in Common Properties of Fields.

Default Bit Value

When the <**set**> field object is default constructed, all bits are initialized to **false**, i.e. **0**. Such default behavior can be modified using **defaultValue** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetField" length="1" defaultValue="true">
            <bit name="SomeBitName" idx="0" />
            <bit name="SomeOtherBitName" idx="1" />
        </set>
    </fields>
</schema>
```

The **SomeSetField** field from the example above is expected to be initialized to **0xff** when default constructed.

The **defaultValue** may also be specified per-bit, which overrides the **defaultValue** specified for the whole field.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetField" length="1" defaultValue="true">
            <bit name="SomeBitName" idx="0" defaultValue="false"/>
            <bit name="SomeOtherBitName" idx="1" />
        </set>
    </fields>
</schema>
```

The **SomeSetField** field from the example above is expected to be initialized to **0xfe** when default constructed.

Reserved Bits

All the bits that aren't listed as <**bit**> XML child elements are considered to be reserved. By default every reserved bit is expected to be zeroed when field is checked to have a valid value. Such

expectation can be changed using **reservedValue** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetField" length="1" defaultValue="true" reservedValue="true">
            <bit name="SomeBitName" idx="0" defaultValue="false" />
            <bit name="SomeOtherBitName" idx="1" defaultValue="false"/>
        </set>
    </fields>
</schema>
```

The **SomeSetField** field from the example above is expected to be initialized to **0xfc** and all the reserved (non-listed) bits are expected to remain **true**.

Reserved bits can also be specified as <**bit**> XML child element with usage of **reserved** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetField" length="1">
            <bit name="SomeBitName" idx="0" />
            <bit name="SomeOtherBitName" idx="1" />
            <bit name="ReservedBit" idx="2" reserved="true">
                <defaultValue value="true" />
                <reservedValue value="true" />
            </bit>
        </set>
    </fields>
</schema>
```

The example above marks bit **2** to be reserved, that is initialized to **true** and must always stay **true**.

The **SomeSetField** field from the example above is expected to be initialized to **0x04** when default constructed.

Endian

The default serialization endian of the protocol is specified in **endian** property of the schema. It is possible to override the default endian value with extra **endian** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <enum name="SomeEnumField" type="uint16" endian="little">
            <bit name="Bit0" idx="0" />
            <bit name="Bit5" idx="5" />
            <bit name="Bit10" idx="10" />
            <bit name="Bit15" idx="15" />
        </enum>
    </fields>
</schema>
```

Allow Non-Unique Bit Names

By default, having multiple names for the same bit is not allowed, the code generator must report an error if two different <**bit**>-s use the same value of **idx** property. It is done as protection against copy-paste errors. However, **CommsDSL** allows usage of multiple names for the same bit in case **nonUniqueAllowed** property has been set to **true**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetField" length="1" nonUniqueAllowed="true">
            <bit name="SomeBitName" idx="0" />
            <bit name="DifferentName" idx="0" />
        </set>
    </fields>
</schema>
```

Versioning

In addition to mentioned earlier properties, every <**bit**> element supports extra ones for versioning:

- **sinceVersion** - Version of the protocol when the bit was introduced (became non-reserved).
- **deprecated** - Version of the protocol when the value was deprecated (became reserved again).

These extra properties are described in detail in Common Properties of Fields.

Version Based Validity

The code generator is expected to generate functionality checking that <**set**> field contains a valid value. Any specified non-reserved bit can have any value, while reserved bits (implicit or explicit) must have value specified by **reservedValue** property (either of the field or the bit itself). By default, the validity check must ignore the version in which particular bit became reserved / non-reserved, and check only values of the bits that have always stayed reserved. However, it is possible to force code generator to generate validity check code that takes into account reported version of

the protocol by using **validCheckVersion** property, which is set to **true**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" version="5">
    <fields>
        <enum name="SomeEnumField" type="uint16" validCheckVersion="true">
            <bit name="Bit0" idx="0" />
            <bit name="Bit5" idx="5" />
            <bit name="Bit10" idx="10" sinceVersion="2" />
            <bit name="Bit15" idx="15" sinceVersion="3" deprecated="4"/>
        </enum>
    </fields>
</schema>
```

In the example above bits **0** and **5** will always have valid values. However bit **10** will be considered valid only if it is cleared before version **2**, and may have any value after. The bit **15** will be allowed any value when version **3** of the protocol is reported, and must be cleared for any other version.

Use properties table for future references.

<bitfield> Field

The <**bitfield**> is a container field, which allows wrapped member fields to be serialized using limited number of bits (instead of bytes). The supported fields, that can be members of the <**bitfield**>, are:

- <enum>
- <int>
- <set>

Since **v2** of the specification it is also allowed to use <ref> field, which references one of the field types above.

The <**bitfield**> field has all the common properties as well as extra properties and elements described below.

Member Fields

Member fields need to be listed as children XML elements of the <**bitfield**>. Every such member is expected to use **bitLength** property to specify its serialization length **in bits**. If it is not specified, then length in bits is calculated automatically as length **in bytes** multiplied by **8**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <bitfield name="SomeBitfield">
            <int name="SomeIntMember" type="uint8" bitLength="3" />
            <set name="SomeSetMember" bitLength="3">
                ...
            </set>
            <enum name="SomeEnumMember" type="uint8" bitLength="2">
                ...
            </enum>
        </bitfield>
    </fields>
</schema>
```

NOTE that summary of all the lengths in bits of all the members must be divisible by **8** and mustn't exceed **64** bits, otherwise the code generator must report an error.

The members of <**bitfield**> must be listed in order starting from the **least** significant bit. In the example above **SomeIntMember** occupies bits [0 - 2], **SomeSetMember** occupies bits [3 - 5], and **SomeEnumMember** occupies bits [6 -7].

If there is any other property defined as XML child of the <**bitfield**>, then all the members must be wrapped in <**members**> XML element for separation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <bitfield name="SomeBitfield">
            <displayName value="Proper Bitfield Name" />
            <members>
                <int name="SomeIntMember" type="uint8" bitLength="3" />
                <set name="SomeSetMember" bitLength="3">
                    ...
                </set>
                <enum name="SomeEnumMember" type="uint8" bitLength="2">
                    ...
                </enum>
            </members>
        </bitfield>
    </fields>
</schema>
```

Endian

When serializing, the <**bitfield**> object needs to combine the values of all the members into single unsigned raw value of appropriate length, and write the received value using appropriate endian. By default **endian** of the schema is used, unless it is overridden using extra **endian** property of the

<**bitfield**> field.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <bitfield name="SomeBitfield" endian="little">
            ...
        </bitfield>
    </fields>
</schema>
```

Use properties table for future references.

<bundle> Field

The <**bundle**> is a container field, which aggregates multiple independent fields (of any kind) into a single one. The <**bundle**> field has all the common properties.

Member Fields

Member fields need to be listed as children XML elements of the <**bundle**>.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <bundle name="SomeBundle">
            <int name="SomeIntMember" type="uint8" />
            <set name="SomeSetMember" length="1">
                ...
            </set>
            <enum name="SomeEnumMember" type="uint8">
                ...
            </enum>
            <bundle name="SomeInnerBundle">
                ...
            </bundle>
        </bundle>
    </fields>
</schema>
```

If there is any other property defined as XML child of the <**bundle**>, then all the members must be wrapped in <**members**> XML element for separation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <bundle name="SomeBundle">
            <displayName value="Proper Bundle Name" />
            <members>
                <int name="SomeIntMember" type="uint8" bitLength="3" />
                <set name="SomeSetMember" length="1">
                    ...
                </set>
                <enum name="SomeEnumMember" type="uint8">
                    ...
                </enum>
                <bundle name="SomeInnerBundle">
                    ...
                </bundle>
            </members>
        </bundle>
    </fields>
</schema>
```

Reusing Other Bundle

Like any other field, <**bundle**> supports **reuse** of any other <**bundle**>. Such reuse copies all the fields from original <**bundle**> in addition to all the properties. Any new defined member field gets **appended** to the copied ones.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <bundle name="SomeBundle">
            <int name="F1" type="uint32" />
            <float name="F2" type="float" />
        </bundle>

        <bundle name="SomeOtherBundle" reuse="SomeBundle">
            <string name="F3" length="16" />
        </bundle>
    </fields>
</schema>
```

In the example above **SomeOtherBundle** has **3** member fields: **F1**, **F2**, and **F3**.

Alias Names to Member Fields

Sometimes an existing member field may be renamed and/or moved. It is possible to create alias names for the fields to keep the old client code being able to compile and work. Please refer to Aliases chapter for more details.

Use properties table for future references.

<string> Field

This field stores and abstracts away value of a text string. The **<string>** field has all the common properties as well as extra properties and elements described below.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="SomeStringField" />
    </fields>
</schema>
```

Such definition of the **<string>** does **NOT** have any limit on the length of the string, and will consume all the available data in the input buffer.

Default Value

The default value of the **<string>** field when constructed can be specified using **defaultValue** property. If not specified, defaults to empty string.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="SomeStringField" defaultValue="hello" />
    </fields>
</schema>
```

Fixed Length

In case the string value needs to be serialized using predefined fixed length, use **length** property to specify the required value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="SomeStringField" length="16" />
    </fields>
</schema>
```

Length Prefix

Many protocols prefix string with its length. The **CommsDSL** allows definition of such prefix using **lengthPrefix** child element, which must define prefix as <int> field.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="SomeStringField">
            <lengthPrefix>
                <int name="LengthPrefix" type="uint8" />
            </lengthPrefix>
        <string>
    </fields>
</schema>
```

In case the prefix field is defined as external field, **CommsDSL** allows usage of **lengthPrefix** as property, value of which contains name of the referenced field.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="StringLengthPrefix" type="uint8" />
        <string name="SomeStringField" lengthPrefix="StringLengthPrefix" />
    </fields>
</schema>
```

The **CommsDSL** also supports **detached** length prefix, when there are several other fields in the <message> or in the <bundle> between the length field and the <**string**>.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <bundle name=SomeBundle">
            <int name="StringLengthPrefixMember" type="uint8" />
            <int name="SomeOtherFieldMember" type="uint8" />
            <string name="SomeStringFieldMember"
lengthPrefix="$StringLengthPrefixMember" />
        </bundle>
    </fields>

    <message name="Msg1" id="1">
        <int name="StringLengthPrefix" type="uint8" />
        <int name="SomeOtherField" type="uint8" />
        <string name="SomeStringField" lengthPrefix="$StringLengthPrefix" />
    </message>
</schema>
```

NOTE, the existence of **$** prefix when specifying **lengthPrefix** value. It indicates that the referenced field is a sibling in the containing <message> or the <bundle> field.

The code generator is expected to take the existence of such detached prefix into account and

generate correct code for various field operations (read, write, etc...).

Zero Termination Suffix

Some protocols may terminate strings with **0** (zero) byte. The **CommsDSL** support such cases with existence of **zeroTermSuffix** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="SomeStringField" zeroTermSuffix="true" />
    </fields>
</schema>
```

NOTE, that **length**, **lengthPrefix** and **zeroTermSuffix** properties are mutually exclusive, i.e. cannot be used together.

Use properties table for future references.

<data> Field

This field stores and abstracts away value of a raw bytes data. The <**data**> field has all the common properties as well as extra properties and elements described below.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <data name="SomeDataField" />
    </fields>
</schema>
```

Such definition of the <**data**> does **NOT** have any limit on the length of the data, and will consume all the available bytes in the input buffer.

Default Value

The default value of the <**data**> field when constructed can be specified using **defaultValue** property. The value of the property must be case-insestive string of hexadecimal values with even number of characters. The allowed ranges of the characters are: [`0' - `9'], and [`a' - `f']. The ' ' (space) character is also allowed for convenient separation of the bytes. If not specified, defaults to empty data.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <data name="SomeDataField" defaultValue="0123 45 67 89 ab cd eF" />
    </fields>
</schema>
```

The example above is expected to create approprate raw data abstracting field, containing 8 bytes: [0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef] when default constructed.

Fixed Length

In case the data value needs to be serialized using predefined fixed length, use **length** property to specify the required value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <data name="SomeDataField" length="16" />
    </fields>
</schema>
```

Length Prefix

Many protocols prefix raw binary data with its length. The **CommsDSL** allows definition of such prefix using **lengthPrefix** child element, which must define prefix as <int> field.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <data name="SomeDataField">
            <lengthPrefix>
                <int name="LengthPrefix" type="uint16" />
            </lengthPrefix>
        <data>
    </fields>
</schema>
```

In case the prefix field is defined as external field, **CommsDSL** allows usage of **lengthPrefix** as property, value of which contains name of the referenced field.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="DataLengthPrefix" type="uint8" />
        <data name="SomeDataField" lengthPrefix="DataLengthPrefix" />
    </fields>
</schema>
```

The **CommsDSL** also supports **detached** length prefix, when there are several other fields in the <message> or in the <bundle> between the length field and the <**data**>.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <bundle name=SomeBundle">
            <int name="DataLengthPrefixMember" type="uint16" />
            <int name="SomeOtherFieldMember" type="uint16" />
            <data name="SomeDataFieldMember" lengthPrefix="$DataLengthPrefixMember" />
        </bundle>
    </fields>

    <message name="Msg1" id="1">
        <int name="DataLengthPrefix" type="uint16" />
        <int name="SomeOtherField" type="uint16" />
        <data name="SomeDataField" lengthPrefix="$DataLengthPrefix" />
    </message>
</schema>
```

NOTE, the existence of **$** prefix when specifying **lengthPrefix** value. It indicates that the referenced field is a sibling in the containing <message> or the <bundle> field.

The code generator is expected to take the existence of such detached prefix into account and generate correct code for various field operations (read, write, etc...).

NOTE, that **length** and **lengthPrefix** properties are mutually exclusive, i.e. cannot be used together.

Use properties table for future references.

<list> Field

This field stores and abstracts away a list of other fields. The <**list**> field has all the common properties as well as extra properties and elements described below.

Element Field of the List

Every <**list**> must specify its element field. The element field can be defined as XML child of the <**list**> definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <list name="SomeListField">
            <int name="Element" type="uint32" />
        </list>
    </fields>
</schema>
```

If a list element contains multiple fields, they must be bundled as members of the <bundle> field.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <list name="SomeListField">
            <bundle name="Element">
                <int name="Mem1" type="uint32" />
                <int name="Mem2" type="int16" />
                <enum name="Mem3" type="uint16">
                    ...
                </enum>
            </bundle>
        </list>
    </fields>
</schema>
```

In case other properties of the <**list**> field are defined as child XML elements, then the element field definition needs to be wrapped by <**element**> XML child.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <list name="SomeListField">
            <displayName value="Some descriptive name" />
            <element>
                <bundle name="Element">
                    ...
                </bundle>
            </element>
        </list>
    </fields>
</schema>
```

The **CommsDSL** also allows reference of externally defined field to be an element using **element** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <bundle name="ExternallyDefinedElement">
            ...
        </bundle>

        <list name="SomeListField" element="ExternallyDefinedElement" />
    </fields>
</schema>
```

Fixed Count

If the defined list must contain predefined number of elements, use **count** property to provide the required information.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <list name="SomeListField" count="8">
            <int name="Element" type="uint32" />
        </list>
    </fields>
</schema>
```

Count Prefix

Most protocols prefix the variable length lists with number of elements that are going to follow, use **countPrefix** child XML element to specify a field that is going to be used as such prefix.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <list name="SomeListField">
            <element>
                <int name="Element" type="uint32" />
            </element>
            <countPrefix>
                <int name="CountPrefix" type="uint16" />
            </countPrefix>
        </list>
    </fields>
</schema>
```

In case the count prefix field is defined as external field, **CommsDSL** allows usage of **countPrefix** as property, value of which contains name of the referenced field.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="ExternalCountPrefix" type="uint16" />

        <list name="SomeListField" countPrefix="ExternalCountPrefix">
            <int name="Element" type="uint32" />
        </list>
    </fields>
</schema>
```

The **CommsDSL** also supports **detached** count prefix, when there are several other fields in the <message> or in the <bundle> between the count field and the <**list**>.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <bundle name=SomeBundle">
            <int name="DataCountPrefixMember" type="uint16" />
            <int name="SomeOtherFieldMember" type="uint16" />
            <list name="SomeDataFieldMember" countPrefix="$DataCountPrefixMember">
                ...
            </list>
        </bundle>
    </fields>

    <message name="Msg1" id="1">
        <int name="DataCountPrefix" type="uint16" />
        <int name="SomeOtherField" type="uint16" />
        <list name="SomeListField" countPrefix="$DataCountPrefix">
            ...
        </list>
    </message>
</schema>
```

NOTE, the existence of **$** prefix when specifying **countPrefix** value. It indicates that the referenced field is a sibling in the containing <message> or the <bundle> field.

The code generator is expected to take the existence of such detached prefix into account and generate correct code for various field operations (read, write, etc...).

Length Prefix

There are protocols that prefix a list with **serialization length** rather than number of elements. In this case use **lengthPrefix** instead of **countPrefix**. The allowed usage scenarios are exactly the same as described above in the Count Prefix section.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <list name="List1">
            <element>
                <int name="Element" type="uint32" />
            </element>
            <lengthPrefix>
                <int name="LengthPrefix" type="uint16" />
            </lengthPrefix>
        </list>

        <int name="ExternalLengthPrefix" type="uint16" />
        <list name="List2" lengthPrefix="ExternalLengthPrefix">
            <int name="Element" type="uint32" />
        </list>
    </fields>

    <message name="Msg1" id="1">
        <int name="DetachedLengthPrefix" type="uint16" />
        <int name="SomeOtherField" type="uint16" />
        <list name="List3" lengthPrefix="$DetachedLengthPrefix">
            ...
        </list>
    </message>
</schema>
```

NOTE, that **count**, **countPrefix** and **lengthPrefix** properties are mutually exclusive, i.e. cannot be used together.

Element Length Prefix

Some protocols prefix every element with its serialization length for the forward / backward compatibility of the protocol. If there is such need, use **elemLengthPrefix** to specify a field that will prefix every element of the list.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <list name="List1">
            <element>
                <bundle name="Element>
                    ...
                </bundle>
            </element>
            <countPrefix>
                <int name="CountPrefix" type="uint16" />
            </countPrefix>
            <elemLengthPrefix>
                <int name="ElemLengthPrefix" type="uint8" />
            </elemLengthPrefix>
        </list>

        <int name="ExternalElemLengthPrefix" type="uint8" />
        <list name="List2" count="16" elemLengthPrefix="ExternalElemLengthPrefix">
            <bundle name="Element>
                ...
            </bundle>
        </list>
    </fields>
</schema>
```

In case every list element has fixed length and protocol specification doesn't allow adding extra variable length fields to the element in the future, some protocols prefix only **first** element in the list with its serialization length. **CommsDSL** supports such lists with **elemFixedLength** property, that has boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <list name="SomeListField" elemFixedLength="true" count="8">
            <element>
                <bundle name="Element">
                    <int name="Mem1" type="uint32" />
                    <int name="Mem2" type="uint32" />
                    ...
                </bundle>
            </element>
            <elemLengthPrefix>
                <int name="ElemLengthPrefix" type="uint8" />
            </elemLenghtPrefix>
        </list>
    </fields>
</schema>
```

The code generator must report an error when element of such list (with **elemFixedLength** property set to **true**) has variable length.

Use properties table for future references.

<float> Field

This field stores and abstracts away value of floating poing type with IEEE 754 encoding. The <**float**> field has all the common properties as well as extra properties and elements described below.

Underlying Type

Every <**float**> field must provide its underlying storage type using **type** property. Available values are:

- **float** - 4 byte floating point representation.
- **double** - double precision 8 bytes floating point representation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="F1" type="float" />
        <float name="F2" type="double" />
    </fields>
</schema>
```

Special Values

Some protocol may set a special meaning for some values. Such values (if exist) must be listed as <**special**> child of the <**float**> XML element.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double">
            <special name="Special1" val="1.0" />
            <special name="Special2" val="5.0" />
        </int>
    </fields>
</schema>
```

The code generator is expected to generate extra convenience functions that check whether field has special value as well as updating the stored value with special one.

Every <**special**> must define a valid name (using **name** property) as well as floating point value (using **val** property), that fits chosen underlying type. The <**special**>-s may be listed in any order,

not necessarily sorted.

In addition to floating point numbers, the **val** property may also cantain the following case-insensitive strings.

- **nan** - Represents NaN value.
- **inf** - Represents positivie infinity.
- **-inf** - Represents negative infinity.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double">
            <special name="Invalid" val="nan" />
        </int>
    </fields>
</schema>
```

Every <**special**> has extra optional properties:

- **description** - Extra description and documentation on how to use the value.
- **sinceVersion** - Version of the protocol when the special name / meaning was introduced.
- **deprecated** - Version of the protocol when the special name / meaning was deprecated.
- **displayName** - Readable name to display for the special value in protocol debugging and visualization tools.

All these extra properties are described in detail in Common Properties of Fields.

By default, non-unqiue special values (different name for the same value) are not allowed, the code generator must report an error if two different <**special**>-es use the same value of the **val** property. It is done as protection against copy-paste errors. However, **CommsDSL** allows usage of non-unique values in case **nonUniqueSpecialsAllowed** property has been set to **true**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloat" type="float" nonUniqueSpecialsAllowed="true">
            <special name="S1" val="0.0" />
            <special name="S2" val="0.0" />
        </int>
    </fields>
</schema>
```

Default Value

The default value of the <**float**> field when constructed can be specified using **defaultValue**

property. If not specified, defaults to **0.0**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double" defaultValue="5.0" />
    </fields>
</schema>
```

The default value can also be specified using the name of one of the <**special**>-s:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double" defaultValue="Special2">
            <special name="Special1" val="nan" />
            <special name="Special2" val="-inf" />
        </float>
    </fields>
</schema>
```

Just like with special values, the value of the **defaultValue** property can also be either **nan**, **inf** or **-inf**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double" defaultValue="nan" />
    </fields>
</schema>
```

Endian

The default serialization endian of the protocol is specified in **endian** property of the schema. It is possible to override the default endian value with extra **endian** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <float name="SomeFloatField" type="double" endian="little" />
    </fields>
</schema>
```

Units

Protocols quite often specify what units are being transfered. The **CommsDSL** provides **units** property to specify this information. The code generator may use this information to generate a functionality that allows retrieval of proper value for requested units, while doing all the conversion math internally. Such behavior will allow developers, that use generated protocol code, to focus on their business logic without getting into details on how value was transfered.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double" units="mm" />
    </fields>
</schema>
```

For list of supported **units** values, refer to appended units table.

Valid Values

Many protocols specify ranges of values the field is allowed to have and how client code is expected to behave on reception of invalid values. The code generator is expected to generate code that checks whether field's value is valid. The **CommsDSL** provides multiple properties to help with such task.

One of such properties is **validRange**. The format of it's value is "[**min_value**, **max_value**]".

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double" validRange="[-100.0, 100.0]" />
    </fields>
</schema>
```

It is possible to have multiple valid ranges for the same field. However XML does NOT allow having multiple attributes with the same name. As the result it is required to put extra valid ranges as <**validRange**> children elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double">
             <validRange value="[-100.0, 100.0]" />
             <validRange value="[250.0, 444.56]" />
        </float>
    </fields>
</schema>
```

Another property is **validValue**, which adds single value (not range) to already defined valid ranges / values. Just like with **validRange**, multiple values need to be added as XML children elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double" validRange="[-100, 100]"
validValue="200.0">
            <validValue value="nan" />
        </float>
    </fields>
</schema>
```

The **validValue** property allows adding special values (**nan**, **inf**, and **-inf**) to available valid values / ranges.

There are also **validMin** and **validMax**, which specify single floating point value and are equivalent to having
`validRange="[provided_min_value, max_value_allowed_by_type]"` and
`validRange="[min_value_allowed_by_type, provided_max_value]"` respectively.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double" validMin="-20.0" />
        <float name="SomeOtherFloatField" type="double" validMax="100.0" />
    </fields>
</schema>
```

The specified valid ranges and values are allowed to intersect. The code generator may warn about such cases and/or unify them to limit number of **if** conditions in the generated code for better performance.

If none of the mentioned above validity related options has been used, the whole range of available values is considered to be valid, including extra values **nan**, **inf**, and **-inf**.

In case **nan**, **inf**, and **-inf** need to be excluded from a range of valid values, but all the available floating point values are considered to be valid, then **validFullRange** property with boolean value needs to be used.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double" validFullRange="true" />
    </fields>
</schema>
```

All the validity related properties mentioned in this section (**validRange**, **validValue**, **validMin**, **validMax**) may also add information about version they were introduced / deprecated in. Adding such information is possible only when the property is defined as XML child element.

```
<?xml version="1.0" encoding="UTF-8"? version="10">
<schema ...>
    <fields>
        <float name="SomeFloatField" type="float">
             <validRange value="[0.0, 10.0]" />
             <validValue value="25.0" sinceVersion="2" deprecated="5" />
             <validRange value="[55.0, 80.0]" sinceVersion="7" />
        </float>
    </fields>
</schema>
```

The **sinceVersion** and **deprecated** properties are described in detail as Common Properties of Fields.

Version Based Validity

The code generator is expected to generate functionality checking that <**float**> field contains a valid value. By default if the field's value is within any of the specified ranges / values, then the it is considered to be valid regardless of version the containing range was introduced and/or deprecated. However, it is possible to force code generator to generate validity check code that takes into account reported version of the protocol by using **validCheckVersion** property, which is set to **true**.

```
<?xml version="1.0" encoding="UTF-8"? version="10">
<schema ...>
    <fields>
        <float name="SomeFloatField" type="float" validCheckVersion="true">
             <validRange value="[0.0, 10.0]" />
             <validValue value="25" sinceVersion="2" deprecated="5" />
             <validRange value="[55, 80]" sinceVersion="7" />
        </float>
    </fields>
</schema>
```

Extra Display Properties

When displaying the floating point value, held by the <**float**> field, GUI analysis tools require knowledge on how many digits after decimal point need to be displayed. To provide this information, use **displayDecimals** property with numeric value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="float" displayDecimals="4" />
    </fields>
</schema>
```

If value of **displayDecimals** is **0**, then it is up to the GUI tool to choose how many digits after decimal point to display.

When <**special**> values are specified the protocol analysis tools are expected to display them next to actual numeric value of the field. The **displaySpecials** property with boolean value is there to control this default behavior.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloat" type="float" displaySpecials="false">
            <special name="S1" val="0.0" />
            <special name="S2" val="nan" />
        </int>
    </fields>
</schema>
```

Use properties table for future references.

<ref> Field

This field serves as reference (alias) to other fields. It can be used to avoid duplication of field definition for multiple messages.

```
<?xml version="1.0" encoding="UTF-8"? version="10">
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint8" defaultValue="Special2">
            <displayName value="Some Int Field" />
            <special name="Special1" val="0" />
            <special name="Special2" val="0xff" />
        </int>
    </fields>

    <message name="Msg1" id="1">
        <ref field="SomeIntField" />
        ...
    </message>

    <message name="Msg2" id="2">
        <ref field="SomeIntField" name="RenamedIntField" />
    </message>
</schema>
```

The **<ref>** field has all the common properties. It also copies **name**, **displayName** and **semanticType** properties from the referenced field and allows overriding them with new values. Note, that in the example above **<ref>** field defined as a member of **Msg1** message hasn't provided any **name** value. It is allowed because it has taken a name of the referenced field (**SomeIntField**).

Referencing the Field

The only extra property the **<ref>** field has is **field** to specify a reference to other field.

Length in Bits

Since **v2** of this specification it is allowed to use **<ref>** field as member of the <bitfield> field while referencing one of the allowed member types. In such case it is required to use **bitLength** property to specify length in bits.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <enum name="SomeEnum" type="uint8">
            <validValue name="V1" val="0" />
            <validValue name="V2" val="1" />
            <validValue name="V3" val="2" />
        </enum>

        <bitfield name="SomeBitfield">
            <int name="SomeIntMember" type="uint8" bitLength="3" />
            <set name="SomeSetMember" bitLength="3">
                ...
            </set>
            <ref field="SomeEnum" bitLength="2" />
        </bitfield>
    </fields>
</schema>
```

Use properties table for future references.

<optional> Field

This field wraps other fields and makes the wrapped field optional, i.e. the serialization of the latter may be skipped if it is marked as "missing". The <**optional**> field has all the common properties as well as extra properties and elements described below.

Inner (Wrapped) Field

Every <**optional**> must specify its inner field. The wrapped field can be defined as XML child of the <**optional**> definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <optional name="SomeOptionalField">
            <int name="SomeOptionalField" type="uint32" />
        </optional>
    </fields>
</schema>
```

In case other properties of the <**optional**> field are defined as child XML elements, then the element field definition needs to be wrapped by <**field**> XML child.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <optional name="SomeOptionalField">
            <displayName value="Some descriptive name" />
            <field>
                <int name="SomeOptionalField" type="uint32" />
            </field>
        </optional>
    </fields>
</schema>
```

The **CommsDSL** also allows reference of externally defined field to be specified as inner (wrapped) field type using **field** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeExternalField" type="uint32" />
        <optional name="SomeOptionalField" field="SomeExternalField" />
    </fields>
</schema>
```

Default Mode

Every <**optional**> field has 3 modes: **tenative** (default), **exist**, and **missing**. The **exist** and **missing** modes are self explanatory. The **tentative** mode is there to perform **read** operation on the inner field only if there are non-consumed bytes left in the input buffer. This mode can be useful with protocols that just add fields at the end of the message in the new version, but the protocol itself doesn't report its version in any other way.

The default mode of the newly constructed <**optional**> field can be specified using **defaultMode** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <optional name="SomeOptionalField" defaultMode="missing">
            <int name="SomeOptionalField" type="uint32" />
        </optional>
    </fields>
</schema>
```

Existence Condition

Many protocols introduce optional fields, and the existence of such fields depend on the value of

some other field. Classic example would be having some kind of flags field (see **<set>**) where some bit specifies whether other field that follows exists or not. Such conditions can be expressed using **cond** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        <set name="Flags">
            <bit name="Val1Exists" idx="0" />
        </set>
        <optional name="Val1" defaultMode="missing" cond="$Flags.Val1Exists">
            <int name="WrappedVal1" type="uint32" />
        </optional>
    </fields>
</schema>
```

NOTE, that **cond** property can be used only for <**optional**> field that is a member of a <bundle> or a <message> . The **cond** expression specifies condition when the <**optional**> field exists, and must always reference other **sibling** field. Such reference is always prefixed with **$** character to indicate that the field is a **sibling** of the <**optional**> and not some external field.

The allowed **cond** expressions are:

- $*set_field_name.bit_name* - The wrapped field exists if specified bit is set to **true** (**1**).
- !$*set_field_name.bit_name* - The wrapped field exists if specified bit is set to **false** (**0**).
- $*field_name compare_op value* - The wrapped field exists if comparison of the specified field with specified value is true. The **compare_op** can be: = (equality), **!**= (inequality), < (less than), <= (less than or equal), > (greater than), >= (greater than or equal).
- $*field_name compare_op* $*other_field_name* - The wrapped field exists if comparison of the specified fields is true.

NOTE, that XML doesn't allow usage of < or > symbols in condition values directly. They need to be substituted with < and > strings respectively.

For example, there are 2 normal fields followed by a third optional one. The latter exists, only if value of **F1** is less than value of **F2**

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        <int name="F1" type="uint16" />
        <int name="F2" type="uint16" />
        <optional name="F3" defaultMode="exists" cond="$F1 &lt; $F2">
            <int name="WrappedF3" type="uint32" />
        </optional>
    </fields>
</schema>
```

Multiple Existence Conditions

The **CommsDSL** also allows usage of multiple existence condition statements. However, they need to be wrapped by either <**and**> or <**or**> XML child elements, which represent "**and**" and "**or**" logical conditions respectively.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        <int name="F1" type="uint16" />
        <int name="F2" type="uint16" />
        <optional name="F3" defaultMode="exists">
            <field>
                <int name="WrappedF3" type="uint32" />
            </field>
            <or>
                <cond value="$F1 = 0" />
                <and>
                    <cond value="$F1 = 1" />
                    <cond value="$F2 != 0" />
                </and>
            </or>
        </optional>
    </fields>
</schema>
```

In the example the **F3** field exists in one of the following conditions:

- Value of **F1** is 0.
- Value of **F1** is 1 and value of **F2** is not 0.

Extra Display Property

By default GUI protocol analysis tools should allow manual update of the <**optional**> field mode. However, if the mode is controlled by the values of other fields, it is possible to disable manual update of the mode by using **displayExtModeCtrl** (stands for "display external mode control") property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        <set name="Flags">
            <bit name="Val1Exists" idx="0" />
        </set>
        <optional name="Val1" defaultMode="missing" cond="$Flags.Val1Exists">
            <displayExtModeCtrl value="true" />
            <field>
                <int name="WrappedVal1" type="uint32" />
            </field>
        </optional>
    </fields>
</schema>
```

Use properties table for future references.

<variant> Field

This field is basically a **union** of other fields. It can hold only one field at a time out of the provided list of supported fields. The <**variant**> field has all the common properties as well as extra properties and elements described below.

Member Fields

The <**variant**> field is there to support heterogeneous lists of fields. The classic example would be a list of **key-value** pairs, where numeric **key** defines what type of **value** follows. Similar to <bundle> field, member fields need to be listed as children XML elements of the <**variant**>.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <int name="Key" type="uint8" failOnInvalid="true" displayReadOnly="true" />

        <variant name="Property">
            <bundle name="Prop1">
                <int reuse="Key" defaultValue="0" validValue="0" />
                <int name="Value" type="uint32" />
            </bundle>
            <bundle name="Prop2">
                <int reuse="Key" defaultValue="1" validValue="1" />
                <string name="Value" length="16" />
            </bundle>
        </variant>

        <list name="PropertiesList" element="Property" />
    </fields>
</schema>
```

The example above defines every **key-value** pair as <bundle> field, where first field (**key**) reuses external definition of **Key** field and adds its default construction value as well as what value considered to be valid. **NOTE**, that every **key** member sets **failOnInvalid** property to **true**. The code generator is expected to generate code that attempts **read** operation of every defined member field in the order of their definition. Once the **read** operation is successful, the right member has been found and the **read** operation needs to terminate. The **in-order** read is high level logic, the code generator is allowed to introduce optimizations as long as the outcome of detecting the right member is the same.

If there is any other property defined as XML child of the <**variant**>, then all the members must be wrapped in <**members**> XML element for separation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <variant name="Property">
            <displayName value="Proper Variant Name" />
            <members>
                <bundle name="Prop1">
                    ...
                </bundle>
                ...
            </members>
        </variant>
    </fields>
</schema>
```

Another quite popular example of is to have heterogeneous list of TLV (type / length / value) triplets.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <int name="Type" type="uint8" failOnInvalid="true" displayReadOnly="true" />

        <variant name="Property">
            <bundle name="Prop1">
                <int reuse="Type" defaultValue="0" validValue="0" />
                <int name="Length" type="uint16" semanticType="length" />
                <int name="Value" type="uint32" />
            </bundle>
            <bundle name="Prop2">
                <int reuse="Type" defaultValue="1" validValue="1" />
                <int name="Length" type="uint16" semanticType="length" />
                <string name="Value" />
            </bundle>
            ...
        </variant>

        <list name="PropertiesList" element="Property" />
    </fields>
</schema>
```

Please **note** assigning **semanticType** property to be **length** for the **Length** field in every bundle. It specifies that the field contains remaining length of all the subsequent fields in the <**bundle**> and allows the code generator to produce correct code. The support for **length** value of **semanticType** has been introduced in **v2** of this specification.

Quite often the developers wonder why there is a need to use **remaining length** information for the fields, length of which is constant and known and compile time. It allows introducing more fields in future versions of the protocol while preserving forward / backward compatibility of the protocol. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" version="2">
    <fields>
        <int name="Type" type="uint8" failOnInvalid="true" displayReadOnly="true" />

        <variant name="Property">
            <bundle name="Prop1">
                <int reuse="Type" defaultValue="0" validValue="0" />
                <int name="Length" type="uint16" semanticType="length" />
                <int name="Value" type="uint32" />
                <int name="Value2" type="int16" sinceVersion="2"/>
            </bundle>
            ...
        </variant>

        <list name="PropertiesList" element="Property" />
    </fields>
</schema>
```

The old version of the protocol code, that is not aware of extra field being added in the new version, will be able to skip over unknown data and read the next property from the correct location.

It also allows safe reception and handling of unexpected (or unknown) properties that could be introduced in the future versions of the protocol while still operating correctly.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <int name="Type" type="uint8" failOnInvalid="true" displayReadOnly="true" />

        <variant name="Property">
            ...
            <bundle name="UnknownProp">
                <int reuse="Type" failOnInvalid="false" />
                <int name="Length" type="uint16" semanticType="length" />
                <data name="Value" />
            </bundle>
        </variant>

        <list name="PropertiesList" element="Property" />
    </fields>
</schema>
```

NOTE, that in the example above the **UnknownProp** is defined to be the last member field of the <**variant**> field and has non-failing read of its **Type** (**failOnInvalid** property has been set to **false**).

Default Member

When <**variant**> field is constructed, it should not hold any field and when serialized, it mustn't produce any output. However, it is possible to specify default member to which the <**variant**> field should be initialized when constructed. To specify such member use **defaultMember** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <variant name="Property" defaultMember="Prop1">
            <bundle name="Prop1">
                ...
            </bundle>
            ...
        </variant>
    </fields>
</schema>
```

The **defaultMember** property may also specify index instead of the member name.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <variant name="Property" defaultMember="0">
            <bundle name="Prop1">
                ...
            </bundle>
            ...
        </variant>
    </fields>
</schema>
```

Negative number as value of **defaultMember** property will force the <**variant**> field not to have a default member.

Extra Display Property

By default GUI protocol analysis tools should display the index of the held member when displaying <**variant**> field in "read only" mode. However, for some occasions this information may be irrelevant. To hide display of index use **displayIdxReadOnlyHidden** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <fields>
        <variant name="Property" displayIdxReadOnlyHidden="true">
            <bundle name="Prop1">
                ...
            </bundle>
            ...
        </variant>
    </fields>
</schema>
```

Use properties table for future references.

Referencing Values of Other Fields

Quite often there is a need to reuse (or reference) some other values already defined and used for some other fields. The **v1** of this specification allowed referencing the external <enum> **validValue** -s only, while **v2** of this specification extends such functionality to other fields as well. In general, when the other field is referenced its **defaultValue** is taken, unless inner value is referenced, such as **validValue** of the <enum> field or **special** value of the <int> field.

Referencing Values Defined in <enum>-s

Any specified <**validValue**> can be referenced by other numeric fields (not only <**enum**>) when specifying numeric value of some property. To reference it, the <**enum**> name must be specified followed by a **.** (dot) and name of the chosen <**validValue**>

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="SomeEnumField" type="uint8" defaultValue="Val2">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="Val3" val="10"/>
        </enum>

        <int name="SomeIntField" type="uint32" defaultValue="SomeEnumField.Val2" />
    </fields>
</schema>
```

In the example above the **defaultValue** of the **SomeIntField** will be **5**.

When <enum> is referenced by its name, its **defaultValue** is taken.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="SomeEnumField" type="uint8" defaultValue="Val2">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="Val3" val="10"/>
        </enum>

        <int name="SomeIntField" type="uint32" defaultValue="SomeEnumField.Val3">
            <special name="S1" val="SomeEnumField" />
            <special name="S2" val="SomeEnumField.Val1" />
        </int>
    </fields>
</schema>
```

In the example above the **defaultValue** of the **SomeIntField** is **10**, the value of the **S1** special is **5** (equals to **defaultValue** of **SomeEnumField**), and value of the **S2** special is **0**.

Floating point fields can also reference values defined in <enum> fields.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="SomeEnumField" type="uint8" defaultValue="Val2">
            <validValue name="Val1" val="0" />
            <validValue name="Val2" val="5" />
            <validValue name="Val3" val="10"/>
        </enum>

        <float name="SomeFloatField" type="double" defaultValue="SomeEnumField.Val1">
            <special name="S1" val="SomeEnumField.Val2" />
        </float>
    </fields>
</schema>
```

Referencing Values Defined in <int>-s

Similar to <enum> the inner value of <int> field can be referenced by other numeric fields.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeIntField" type="uint32" defaultValue="10">
            <special name="S1" val="0" />
            <special name="S2" val="5" />
        </int>

        <enum name="SomeEnumField" type="uint8" defaultValue="SomeIntField.S1">
            <validValue name="Val1" val="SomeIntField.S2" />
            <validValue name="Val2" val="SomeIntField" />
        </enum>

        <float name="SomeFloatField" type="double" defaultValue="SomeIntField.S2">
            <special name="S1" val="SomeIntField" />
        </float>
    </fields>
</schema>
```

In the example above **defaultValue** of **SomeEnumField** is **0**, **validValue Val1** equals to **5**, and **validValue Val2** equals to **10**.

Also the **defaultValue** of **SomeFloatField** is **5.0**, while value of its **S1** special is **10.0**.

Referencing Values Defined in <set>-s

The **defaultValue** property of any element of the <set> field can also be referenced.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetField" length="1" defaultValue="false">
            <bit name="B0" idx="0" defaultValue="true" />
            <bin name="B1" idx="1" />
        </set>

        <set name="SomeOtherSetField" length="1" defaultValue="SomeSetField"
reservedValue="SomeSetField.B0">
            <bit name="B5" idx="5" defaultValue="SomeSetField.B1" />
            ...
        </set>
    </fields>
</schema>
```

In the example above the **defaultValue** of **SomeOtherSetField** is **false** (same as **defaultValue** of **SomeSetField**), the **reservedValue** of **SetOtherField** is **true** (same as **defaultValue** of **SomeSetField.B0**), and the **defaultValue** of **SomeOtherSetField.B5" is false (same as defaultValue of *SomeSetField.B1**).

Other numeric fields, such as <enum> , <int> , and <float> can also reference boolean values of <set> , which will result in numeric values been either **0** or **1**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <set name="SomeSetField" length="1" defaultValue="false">
            <bit name="B0" idx="0" defaultValue="true" />
            <bin name="B1" idx="1" />
        </set>

        <float name="SomeFloatField" type="float" defaultValue="SomeSetField.B0" />
    </fields>
</schema>
```

The definition above will result in **defaultValue** of **SomeFloatField** to be **1.0**.

Referencing Values Defined in <float>-s

Similar to <int> it is possible to reference <float> values used in **defaultValue** property and/or as <**special**> value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <float name="SomeFloatField" type="double" defaultValue="nan">
            <special name="S1" val="inf" />
        </float>

        <float name="SomeOtherFloatField" type="double"
defaultValue="SomeFloatField.S1">
            <special name="S1" val="SomeFloatField" />
        </float>

    </fields>
</schema>
```

In the example above **defaultValue** of **SomeOtherFloatField** is **inf**, while value of **SomeOtherFloatField.S1** special is **nan**.

Referencing Values Defined in <string>-s

When referencing values of <string> fields there is a need to differentiate between a reference to external field and a genuine string value. To do so the ^ prefix was introduced. If a property value, that requires a string, starts with ^ it means external reference and error must be reported if referenced field is not found.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="SomeString" defaultValue="hello" />
        <string name="SomeOtherString" defaultValue="^SomeString" />
    </fields>
</schema>
```

In the example above the **defaultValue** of **SomeOtherString** field is **hello**.

If there is a need to define a genuine string value that starts with ^ character, then there is a need to escape it with \.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="SomeOtherString" defaultValue="\^SomeString" />
    </fields>
</schema>
```

In the example above the **defaultValue** of **SomeOtherString** field is **^SomeString**.

The question may arise what if a genuine value string needs to start with \^. In this case just add additional \ at the front.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="SomeOtherString" defaultValue="\\^SomeString" />
    </fields>
</schema>
```

In the example above the **defaultValue** of **SomeOtherString** field is **\^SomeString**.

The bottom line: any **prefix** sequence of \ followed by the ^ will result in drop of one \ in the final string value. In case there is any other character used in the middle, the string value remains as is.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="String1" defaultValue="\\SomeString" />
        <string name="String2" defaultValue="\.\^SomeString" />
    </fields>
</schema>
```

In the example above the **defaultValue** of **String1** field is **\\SomeString** because there is no ^

character after \ and the **defaultValue** of **String2** field is \.\^**SomeString** because the sequence of \ is interrupted by ..

NOTE, that string referenced can be useful when <enum> field is used to specify numeric message IDs.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="Msg1Name" defaultValue="Message 1" />
        <string name="Msg2Name" defaultValue="Message 2" />

        <enum name="MsgId" type="uint8" semanticType="messageId">
            <validValue name="Msg1" val="1" displayName="^Msg1Name" />
            <validValue name="Msg2" val="2" displayName="^Msg2Name" />
        </enum>

        <message name="Msg1" id="MsgId.Msg1" displayName="^Msg1Name">
            ...
        </message>

        <message name="Msg2" id="MsgId.Msg2" displayName="^Msg2Name">
            ...
        </message>
    </fields>
</schema>
```

In the example above the **displayName** property of a message is expected to be the same as **displayName** property of appropriate <**validValue**> of **MsgId** enum. Referencing common value insures that the change to the name (if happens) propagates to appropriate fields.

Referencing Values Defined in <data>-s

When referencing values of <data> fields there is also a need to differentiate between a reference to external field and a genuine data value. For example the string `abcd` can be interpreted as valid field name as well as valid hexadecimal bytes. As the result there is also a need to use ^ prefix (just like with <**string**> values) to indicate external reference.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <data name="SomeData" defaultValue="12 34 56" />
        <string name="SomeOtherData" defaultValue="^SomeData" />
    </fields>
</schema>
```

The **defaultValue** of **SomeOtherData** will be `0x12 0x34 0x56`.

Referencing Values via <ref>-s

The <ref> field is there to create an alias to other field. The **CommsDSL** allows retrieving value for the <ref> field as if it was retrieved from the referenced field.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeInt" type="uint8" defaultValue="1">
            <special name="S1" val="5" />
        </int>
        <ref name="SomeRef" field="SomeInt" />
        <int name="SomeOtherInt" type="uint18" defaultValue="SomeRef.S1" />
    </fields>
</schema>
```

In the example above the **defaultValue** of **SomeOtherInt** is **5**, same as the value of **SomeInt.S1**.

Referencing Values in Namespaces

In case referenced field resides in a namespace, add it to the reference string as well. The same referencing rules apply.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <ns name="ns1"
        <fields>
            <enum name="SomeEnumField" type="uint8">
                <validValue name="Val1" val="0" />
                <validValue name="Val2" val="5" />
                <validValue name="Val3" val="10"/>
            </enum>

            <int name="SomeIntField" type="uint32"
defaultValue="ns1.SomeEnumField.Val2" />
        </fields>
    </ns>
</schema>
```

Referencing Values in <bitfield>-s or <bundle>-s

The **CommsDSL** also allows referencing values from member fields of a <bitfield> or a <bundle> .

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <ns name="ns1"
        <fields>
            <bundle name="SomeBundle">
                <enum name="SomeEnumMember" type="uint8">
                    <validValue name="Val1" val="0" />
                    <validValue name="Val2" val="5" />
                    <validValue name="Val3" val="10"/>
                </enum>
                <int name="SomeIntMember" type="uint8" />
            </bundle>

            <int name="SomeIntField" type="uint32"
defaultValue="ns1.SomeBundle.SomeEnumField.Val2" />
        </fields>
    </ns>
</schema>
```

In the example above the **defaultValue** of **SomeIntField** is **5**.

Referencing Values in <optional>-s;

There are two forms of <optional> fields. One references external field, another defines it as a member.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeInt" type="uint8" defaultValue="1">
            <special name="S1" val="5" />
        </int>
        <optional name="Opt1" field="SomeInt" />
        <optional name="Opt2">
            <int name="SomeOptInt" type="uint8" defaultValue="1">
                <special name="S1" val="5" />
            </int>
        </optional>
    </fields>
</schema>
```

In case the <optional> field references external field it can **NOT** be used for value reference. The one that defines optional field internally as a child element, can.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <optional name="Opt2">
            <int name="SomeOptInt" type="uint8" defaultValue="1">
                <special name="S1" val="5" />
            </int>
        </optional>
        <int name="SomeOtherInt" type="int16" defaultValue="Opt2.SomeOptInt.S1" />
    </fields>
</schema>
```

NOTE that there is a need to reference internal member field by name.

Messages

Every message is defined using <**message**> XML element.

Message Name

Every message definition must specify its name using **name** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" ...>
        ...
    </message>
</schema>
```

Numeric ID

Every message definition must specify its numeric ID using **id** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="0x1">
        ...
    </message>
</schema>
```

It is highly recommended to define "message ID" numeric values as external <enum> field and reuse its values.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="MsgId" type="uint8" semanticType="messageId">
            <validValue name="Msg1" val="0x1" />
            <validValue name="Msg2" val="0x2" />
        </enum>
    </fields>

    <message name="Msg1" id="MsgId.Msg1">
        ...
    </message>

    <message name="Msg2" id="MsgId.Msg2">
        ...
    </message>
</schema>
```

Description

It is possible to provide a description of the message about what it is and how it is expected to be used. This description is only for documentation purposes and may find it's way into the generated code as a comment for the generated class. The property is **description**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        <description>
            Some long
            multiline
            description
        </description>
        ...
    </message>
</schema>
```

Display Name

When various analysis tools display message details, the preference is to display proper space separated name (which is defined using **displayName** property) rather than using a name.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1" displayName="Message 1">
        ...
    </message>
</schema>
```

In case **displayName** is empty, the analysis tools are expected to use value of **name** property instead.

It is recommended to share the **displayName** with relevant <**validValue**> of <**enum**> that lists numeric IDs of the messages:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <string name="Msg1Name" defaultValue="Message 1" />
        <string name="Msg2Name" defaultValue="Message 2" />

        <enum name="MsgId" type="uint8" semanticType="messageId">
            <validValue name="Msg1" val="1" displayName="^Msg1Name" />
            <validValue name="Msg2" val="2" displayName="^Msg2Name" />
        </enum>

        <message name="Msg1" id="MsgId.Msg1" displayName="^Msg1Name">
            ...
        </message>

        <message name="Msg2" id="MsgId.Msg2" displayName="^Msg2Name">
            ...
        </message>
    </fields>
</schema>
```

Fields

Every <**message**> has zero or more fields that can be specified as child XML elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <int name="SomeCommonField" type="uint16" defaultValue="10" />
    </fields>

    <message name="Msg1" id="1">
        <int name="F1" type="uint8" />
        <enum name="F2" type="uint8">
            ...
        </enum>
        <ref field="SomeCommonField" name="F3" />
        <string name="F4" length="8" />
        ...
    </message>
</schema>
```

If there is any other property defined as XML child of the <**message**>, then all the fields must be wrapped in <**fields**> XML element for separation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <message name="Msg1" id="1">
        <displayName value="Message 1" />
        <fields>
            <int name="F1" type="uint8" />
        </fields>
    </message>
</schema>
```

Sometimes different messages have the same fields. In order to avoid duplication, use **copyFieldsFrom** property to specify original message.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        <int name="F1" type="uint32" />
    </message>

    <message name="Msg2" id="2" copyFieldsFrom="Msg1" />
</schema>
```

In the example above **Msg2** will have the same fields as **Msg1**.

After copying fields from other message, all other defined fields will be appended to copied ones.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        <int name="F1" type="uint32" />
    </message>

    <message name="Msg2" id="2" copyFieldsFrom="Msg1">
        <float name="F2" type="float" />
    </message>
</schema>
```

In the example above **Msg2** will have 2 fields: **F1** and **F2**.

Ordering

There are protocols that may define various forms of the same message, which share the same numeric ID, but are differentiated by a serialization length or value of some particular field inside the message. It can be convenient to define such variants as separate classes. In case there are multiple <**message**>-es with the same numeric ID, it is required to specify order in which they are expected to be processed (read). The ordering is specified using **order** property with unsigned numeric value. The message object with lower **order** value gets created and its **read** operation attempted **before** message object with higher value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" nonUniqueMsgIdAllowed="true">
    <message name="Msg1Form1" id="1" order="0" >
        ...
    </message>

    <message name="Msg1Form2" id="1" order="1">
        ...
    </message>
</schema>
```

NOTE that there is a need to set **nonUniqueMsgIdAllowed** property of the schema to **true** to allow multiple message objects with the same numeric ID.

All the **order** values for the same numeric ID must be unique, but not necessarily sequential.

Versioning

CommsDSL allows providing an information in what version the message was added to the protocol, as well as in what version it was deprecated, and whether it was removed (not supported any more) after deprecation.

To specify the version in which message was introduced, use **sinceVersion** property. To specify the version in which the message was deprecated, use **deprecated** property. To specify whether the

message was removed after being deprecated use **removed** property in addition to **deprecated**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big" version="5" >
    <message name="SomeMessage" id="100" sinceVersion="2" >
        ...
    </message>

    <message name="SomeOtherMessage" id="101" sinceVersion="3" deprecated="4"
removed="true">
        ...
    </message>

</schema>
```

In the example above **SomeMessage** was introduced in version **2**, and **SomeOtherMessage** was introduced in version **3**, but deprecated and removed in version **4**.

NOTE, that all the specified versions mustn't be greater that the version of the schema. Also value of **sinceVersion** must be **less** than value of **deprecated**.

The code generator is expected to be able to generate support for specific versions and include / exclude support for some messages based on their version information.

Platforms

Some protocols may be used in multiple independent platforms, while having some platform-specific messages. The **CommsDSL** allows listing of the supported platforms using <platform> XML nodes. Every message may list platforms in which it must be supported using **platforms** property. In case the property's value is empty (default), the message is supported in **all** the available platforms (if any defined). The **platforms** property value is coma-separated list of platform names with preceding + if the listed platforms are the one supported, or - if the listed platforms need to be **excluded** from all available ones.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <platform name="Plat1" />
    <platform name="Plat2" />
    <platform name="Plat3" />
    <platform name="Plat4" />

    <message name="Msg1" id="1" platforms="+Plat1,Plat4">
        ...
    </message>

    <message name="Msg2" id="2" platforms="-Plat1, Plat2">
        ...
    </message>
</schema>
```

In the example above **Msg1** is supported only for platforms **Plat1** and **Plat4**, while **Msg2** is **NOT** supported in **Plat1**, and **Plat2** (i.e. supported in **Plat3** and **Plat4**).

The main consideration for what format to choose should be whether the platforms support for the message should or should **NOT** be added automatically when new <**platform**> is defined.

Sender

In most protocols there are uni-directional messages. The **CommsDSL** allows definition of entity that sends a particular message using **sender** property. Available values are **both** (default), **server**, and **client**. The code generator may use provided information and generate some auxiliary code and/or data structures to be used for **client** and/or **server** implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1" sender="client">
        ...
    </message>

    <message name="Msg2" id="2" sender="server">
        ...
    </message>

    <message name="Msg3" id="2">
        ...
    </message>
</schema>
```

In the example above **Msg1** and **Msg2** are uni-directional messages, while **Msg3** is bi-directional.

Customization

The code generator is expected to allow some level of compile time customization of the generated code, such as enable/disable generation of particular virtual functions. The code generator is also expected to provide command line options to choose required level of customization. Sometimes it may be required to allow generated message class to be customizable regardless of the customization level requested from the code generator. **CommsDSL** provides **customizable** property with boolean value to force any message to being customizable at compile time.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1" customizable="true">
        ...
    </message>
</schema>
```

Alias Names to Member Fields

Sometimes an existing member field may be renamed and/or moved. It is possible to create alias names for the fields to keep the old client code being able to compile and work. Please refer to Aliases chapter for more details.

Use properties table for future references.

Interfaces

There are protocols that attach some extra information, such as version and/or extra flags, to the message transport framing. This extra information usually influences how message fields are deserialized and/or how message object is handled. It means that these received extra values need to be attached to **every** message object. The **CommsDSL** allows specification of such extra information as <**interface**> XML element with extra fields.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <interface name="CommonInterface">
        <int name="Version" type="uint8" semanticType="version" />
        <set name="Flags" length="1">
            ...
        </set>
    </interface>
</schema>
```

The code generator may use provided information to generate common interface class(es) for all the messages. Such class will serve as base class of every message object and will contain required extra information.

NOTE that specified fields, are **NOT** part of every message's payload. These fields are there to hold extra values delivered as part of message framing.

Interface Name

Every interface definition must specify its name using **name** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <interface name="CommonInterface">
        ...
    </interface>
</schema>
```

Description

It is possible to provide a description of the interface about what it is and how it is expected to be used. This description is only for documentation purposes and may find it's way into the generated code as a comment for the generated class. The property is **description**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <interface name="CommonInterface">
        <description>
            Some long
            multiline
            description
        </description>
        ...
    </interface>
</schema>
```

More About Fields

Similar to <**message**> every <**interface**> has zero or more fields that can be specified as child XML elements. If there is any other property defined as XML child of the <**interface**>, then all the fields must be wrapped in <**fields**> XML element for separation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" endian="big">
    <interface>
        <name value="CommonInterface" />
        <fields>
            <int name="Version" type="uint8" semanticType="version" />
        </fields>
    </message>
</schema>
```

Sometimes different interfaces have common set of fields. In order to avoid duplication, use **copyFieldsFrom** property to specify original interface and then add extra fields when needed.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <interface name="Interface1">
        <int name="Version" type="uint8" semanticType="version" />
    </interface>

    <interface name="Interface2" copyFieldsFrom="Interface1">
        <set name="Flags" length="1">
            ...
        </set>
    </interface>
</schema>
```

In the example above **Interface2** will have **2** fields: "Version" and "Flags".

If the protocol doesn't have any extra values delivered in message framing, then the whole definition of the <**interface**> XML element can be omitted. If no <**interface**> node has been added to the protocol schema, then the code generator must treat schema as if the schema has implicit definition of the single <**interface**> with no fields. It is up to the code generator to choose name for the common interface class it creates.

NOTE usage of **semanticType**="**version**" for the field holding protocol version in the examples above. It is required to be used for proper protocol versioning. The value of the field having "version" value as **semanticType** property, will be considered for fields that, were introduced and/or deprecated at some stage, i.e. use **sinceVersion** and/or **derecated** + **removed** properties.

Alias Names to Fields

Sometimes a contained field may be renamed and/or moved. It is possible to create alias names for the fields to keep the old client code being able to compile and work. Please refer to Aliases chapter for more details.

Use properties table for future references.

Aliases

It is not uncommon for a particular field to change its meaning and as the result to change its name over time when the protocol evolves. Simple change of the name in the schema may result in various compilation errors of old client code when new version of the protocol definition library is released. To help with such case the **CommsDSL** introduces an ability to create **alias** names for the existing fields.

For example let's assume there is some message definition like the one below:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="0x1">
        <int name="SomeField" type="uint32" />
        ...
    </message>
</schema>
```

In case there is a need to rename the **SomeField** name to be **SomeOtherField**, then the message definition can add an <**alias**> with the old name to the renamed field in order to keep the old client code compiling.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="0x1">
        <fields>
            <int name="SomeOtherField" type="uint32" />
            ...
        </fields>
        <alias name="SomeField" field="$SomeOtherField" />
    </message>
</schema>
```

In such case the code generator must allow access of the renamed field by both old and new names.

Note that the message fields must be bundled in <**fields**> XML element in order to allow usage of non-field definition <**alias**> XML child of the <**message**> node.

Also note that value of the **field** property of the <**alias**> element must start with `$` character to indicate that the referenced field is a sibling one, similar to <optional> field conditions.

Quite often, in order to keep protocol backward compatible, developers convert existing numeric field into a <bitfield> when need arises to add some extra field to the message. For example, let's assume there was an enum field with limited number of valid values:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="0x1">
        <enum name="SomeEnum" type="uint8">
            <validValue name="V1" val="0" />
            <validValue name="V2" val="1" />
            <validValue name="V3" val="2" />
        </enum>
        ...
    </message>
</schema>
```

When need arises to introduce new value the developer may decide to save I/O traffic reuse the same byte occupied by the SomeEnum field, like below.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="0x1">
        <bitfield name="F1"
            <enum name="SomeEnum" type="uint8" bitLength="2">
                <validValue name="V1" val="0" />
                <validValue name="V2" val="1" />
                <validValue name="V3" val="2" />
            </enum>
            <int name="SomeInt" type="uint8" bitLength="6"
        </bitfield>
        ...
    </message>
</schema>
```

In order to keep old client code compiling it is possible to introduce alias to the SomeEnum member of the <bitfield> like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="0x1">
        <fields>
            <bitfield name="F1"
                <enum name="SomeEnum" type="uint8" bitLength="2">
                    ...
                </enum>
                <int name="SomeInt" type="uint8" bitLength="6"
            </bitfield>
            ...
        </fields>
        <alias name="SomeEnum" field="$F1.SomeEnum" />
    </message>
</schema>
```

There can be any number of different **<alias>** nodes. The elements that are allowed to have **<alias>** -es are <message> , <interface> , and <bundle> .

Description

The **<alias>** node may also have **description** property which is expected to find its way into the generated code as a comment for the relevant access functions.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        ...
        <alias name="SomeField" field="SomeOtherField">
            <description>
                Some long
                multiline
                description
            </description>
        </alias>
    </message>
</schema>
```

More on Aliases in <message>-es

When a new <message> is defined it can copy all the fields from other already defined <message> (using **copyFieldsFrom** property). By default all the <alias> definitions are also copied. It is possible to modify this default behavior by using **copyFieldsAliases** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <message name="Msg1" id="1">
        ...
    </message>

    <message name="Msg2" id="2" copyFieldsAliases="false">
        ...
    </message>

</schema>
```

More on Aliases in <interface>-es

Similar to <message> -es <interface> can also use **copyFieldsFrom** property to copy its field from some other <interface> definition and have all the aliases copied by default. The control of such default copying behavior is also done by using **copyFieldsAliases** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <interface name="Interface1">
        ...
    </interface>

    <interface name="Interface2" copyFieldsAliases="false">
        ...
    </interface>

</schema>
```

More on Aliases in <bundle>-es

When a new <bundle> field is defined it can reuse definition of already defined other <bundle> (using **reuse** property). By default all the <alias> definitions are also copied. It is possible to modify this default behavior by using **reuseAliases** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <bundle name="B1">
            ...
            <alias .../>
            <alias .../>
        </bundle>

        <bundle name="B2" reuse="B1" reuseAliases="false">
            ...
        </bundle>
    </fields>
</schema>
```

Use properties table for future references.

Frames

Every communication protocol must ensure that the message is successfully delivered over the I/O link to the other side. The serialised message payload must be wrapped in some kind of transport information prior to being sent and unwrapped on the other side when received. The **CommsDSL** allows specification of such transport wraping using <**frame**> XML node.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame ...>
        ...
    </frame>
</schema>
```

Name

Every frame definition must specify its name using **name** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        ...
    </frame>
</schema>
```

Description

It is possible to provide a description of the frame about what it is and how it is expected to be used. This description is only for documentation purposes and may find it's way into the generated code as a comment for the generated class. The property is **description**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <description>
            Some long
            multiline
            description
        </description>
        ...
    </frame>
</schema>
```

Layers

The protocol framing is defined using so called "layers", which are additional abstraction on top of fields, where every such layer has a specific purpose. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="MsgId" type="uint8" semanticType="messageId">
            <validValue name="Msg1" val="0x1" />
            <validValue name="Msg2" val="0x2" />
        </enum>
    </fields>

    <frame name="ProtocolFrame">
        <id name="Id" field="MsgId" />
        <payload name="Data" />
    </frame>
</schema>
```

The example above defines simple protocol framing where 1 byte of numeric message ID precedes the message payload.

```
ID (1 byte) | PAYLOAD
```

Available layers are:

- <payload> - Message payload.

- <id> - Numeric message ID.
- <size> - Remaining size (length).
- <sync> - Synchronization bytes.
- <checksum> - Checksum.
- <value> - Extra value, usually to be assigned to one of the <interface> fields.
- <custom> - Any other custom layer, not defined by **CommsDSL**.

If there is any other property defined as XML child of the <**frame**>, then all the layers must be wrapped in <**layers**> XML element for separation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame>
        <name value="ProtocolFrame" />
        <layers>
            <id name="Id" field="MsgId" />
            <payload name="Data" />
        </layers>
    </frame>
</schema>
```

All these layers have common as well as their own specific set of properties.

Use properties table for future references.

Common Properties of Layers

Every layer is different, and defines its own properties and/or other aspects. However, there are common properties, that are applicable to every layer. They are summarized below.

Name

Every layer must define it's name, which is expected to be used by a code generator when defining a relevant class. The name is defined using **name** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <id name="Id" ... />
        <payload name="Data" />
    </frame>
</schema>
```

Description

It is possible to provide a description of the layer about what it is and how it is expected to be used. This description is only for documentation purposes and may find it's way into the generated code as a comment for the generated class. The property is **description**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <id name="Id" ...>
            <description>
                Some long
                multiline
                description
            </description>
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

Inner Field

Every layer, except for <payload> , needs to specify its inner fields it wraps. The field can be specified as XML child element.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <id name="Id">
            <int name="IdField" type="uint8" />
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

If there is any other property defined as XML child of the layer, then the inner field must be wrapped in <**field**> XML element for separation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <id>
            <name value="Id" />
            <field>
                <int name="IdField" type="uint8" />
            </field>
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

If the inner field is defined globally in <**fields**> section, it can be referenced using **field** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <fields>
        <enum name="MsgId" type="uint8" semanticType="messageId">
            <validValue name="Msg1" val="0x1" />
            <validValue name="Msg2" val="0x2" />
        </enum>
    </fields>

    <frame name="ProtocolFrame">
        <id name="Id" field="MsgId" />
        <payload name="Data" />
    </frame>
</schema>
```

Use properties table for future references.

<payload> Layer

The <**payload**> layer represents message payload. It is the only **must-have** layer in the frame definition. It doesn't have any extra properties in addition to common ones.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        ...
        <payload name="Data">
    </frame>
</schema>
```

<id> Layer

The <**id**> layer represents numeric message ID. The frame definition must **NOT** contain more than one <**id**> layer.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <id name="Id">
            <int name="IdField" type="uint8" />
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

The <**id**> layer doesn't have any extra properties in addition to common ones.

<size> Layer

The <**size**> layer represents **remaining** serialization length of the frame until the end of <payload> . The frame definition must **NOT** contain more than one <**size**> layer.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <size name="Size">
            <int name="SizeField" type="uint16" />
        </size>
        <id name="Id">
            <int name="IdField" type="uint8" />
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

Some protocols may specify that the field of the <**size**> layer contains its own length as well. The **CommsDSL** allows implementation of such case by adding usage of **serOffset** property to the field of the <**size**> layer.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <size name="Size">
            <int name="SizeField" type="uint16" serOffset="2" displayOffset="2"/>
        </size>
        <id name="Id">
            <int name="IdField" type="uint8" />
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

The example below implements `ID (2 bytes) | SIZE (2 bytes) | PAYLOAD` framing where `SIZE` value includes length of the header (`ID` + `SIZE`) in addition to the length of `PAYLOAD`.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <id name="Id">
            <int name="IdField" type="uint16" />
        </id>
        <size name="Size">
            <int name="SizeField" type="uint16" serOffset="4" displayOffset="4"/>
        </size>
        <payload name="Data" />
    </frame>
</schema>
```

Also **NOTE** that **<size>** layer specifies number of **remaining** bytes **until the end of <payload> layer**. There are protocols that append some kind of <checksum> after the payload. In order to include them in the value of the **<size>** layer, also use **serOffset** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
    <frame name="ProtocolFrame">
        <size name="Size">
            <int name="SizeField" type="uint16" serOffset="2" displayOffset="2"/>
        </size>
        <id name="Id">
            <int name="IdField" type="uint8" />
        </id>
        <payload name="Data" />
        <checksum ...>
            <int name="ChecksumField" type="uint16" />
        </checksum>
    </frame>
</schema>
```

The <**size**> layer doesn't have any extra properties in addition to common ones.

<sync> Layer

The <**sync**> layer represents synchronization bytes, usually (but not always) present at the beginning of the frame.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        <sync name="Sync">
            <int name="SyncField" type="uint16" defaultValue="0xabcd">
                <validValue value="0xabcd" />
                <failOnInvalid value="true" />
            </int>
        </sync>
        <size name="Size">
            <int name="SizeField" type="uint16" />
        </size>
        <id name="Id">
            <int name="IdField" type="uint8" />
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

The example below implements `SYNC (2 bytes) | ID (2 bytes) | SIZE (2 bytes) | PAYLOAD` framing where `SYNC` must be `0xab 0xcd` bytes. Note, that read of the `SyncField` will fail in case its read value is not `0xabcd`.

The <**sync**> layer doesn't have any extra properties in addition to common ones.

<checksum> Layer

The <**checksum**> layer represents checksum bytes, usually (but not always) present at the end of the frame.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        <sync name="Sync">
            ...
        </sync>
        <size name="Size">
            ...
        </size>
        <id name="Id">
            ...
        </id>
        <payload name="Data" />
        <checksum name="Checksum" ...>
            <int name="ChecksumField" type="uint16" />
        </checksum>
    </frame>
</schema>
```

The <**checksum**> layer has all the common properties as well as extra properties and elements described below.

Checksum Algorithm

The checksum calculation algorithm must be specified using **alg** property. Supported values are:

- **sum** - Sum of all the bytes.
- **crc-ccitt** (aliases: **crc_ccitt**) - CRC-16-CCITT where polynomial is `0x1021`, initial value is `0xffff`, final XOR value is `0`, and no reflection of bytes.
- **crc-16** (aliases: **crc_16**) - CRC-16-IBM, where polynomial is `0x8005`, initial value is `0`, final XOR value is `0`, reflection is performed on every byte as well as final value.
- **crc-32** (aliases: **crc_32**) - CRC-32, where polynomial is `0x04C11DB7`, initial value is `0xffffffff`, final XOR value is `0xffffffff`, reflection is performed on every byte as well as final value.
- **custom** - Custom algorithm, code for which needs to be provided to the code generator, so it can copy it to the generated code.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        ...
        <payload name="Data" />
        <checksum name="Checksum" alg="crc-16" ...>
            <int name="ChecksumField" type="uint16" />
        </checksum>
    </frame>
</schema>
```

When **custom** algorithm is selected, its name must be provided using **algName** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        ...
        <payload name="Data" />
        <checksum name="Checksum" alg="custom" algName="MyCustomChecksumCalc" ...>
            <int name="ChecksumField" type="uint16" />
        </checksum>
    </frame>
</schema>
```

The provided name of the custom algorithm can be used by the code generator to locate the required external implementation file and use appropriate class / function name when the calculation functionality needs to be invoked.

Calculation Area

The **checksum** layer definition must also specify the layers, data of which is used to calculate the checksum. It is done using **from** property that is expected to specify name of the layer where checksum calculation starts.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        <sync name="Sync">
            ...
        </sync>
        <size name="Size">
            ...
        </size>
        <id name="Id">
            ...
        </id>
        <payload name="Data" />
        <checksum name="Checksum" alg="crc-16" from="Size">
            <int name="ChecksumField" type="uint16" />
        </checksum>
    </frame>
</schema>
```

The example above defines `SYNC | SIZE | ID | PAYLOAD | CHECKSUM` frame where the checksum is calculated on `SIZE` + `ID` + `PAYLOAD` bytes.

Some protocols may put checksum value as prefix to the area on which the checksum needs to be calculated. In this case use **until** property (instead of **from**) to specify layers for checksum calculation.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        <sync name="Sync">
            ...
        </sync>
        <size name="Size">
            ...
        </size>
        <id name="Id">
            ...
        </id>
        <checksum name="Checksum" alg="crc-16" until="Data">
            <int name="ChecksumField" type="uint16" />
        </checksum>
        <payload name="Data" />
    </frame>
</schema>
```

Checksum Verification Order

The default behavior of the <**checksum**> layer is to perform calculation after all relevant layers and

their fields have been successfully read and processed. However, it is possible to force the checksum verification right away (without reading fields of other layers and/or message payload). It is usually possible to do when value of the <size> field is already processed, so the right location of checksum value is known, and it is not included in checksum calculation. To force immediate checksum verification use **verifyBeforeRead** property with boolean value.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        <sync name="Sync">
            ...
        </sync>
        <size name="Size">
            ...
        </size>
        <id name="Id">
            ...
        </id>
        <checksum name="Checksum" alg="crc-16" until="Data" verifyBeforeRead="true">
            <int name="ChecksumField" type="uint16" />
        </checksum>
        <payload name="Data" />
    </frame>
</schema>
```

Use properties table for future references.

<value> Layer

The <**value**> layer represents extra values (such as protocol version and/or extra flags), that are applicable to all the messages, but delivered as part of transport framing instead of message payload.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        <size name="Size">
            ...
        </size>
        <value name="Version" ...>
            <int name="VersionField" type="uint8" />
        </value>
        <id name="Id">
            ...
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

The <**value**> layer has all the common properties as well as extra properties and elements described below.

Interfaces

In most cases the received value must be reassigned to appropriate field of the <interface> . To specify the supported interfaces use **interfaces** property. The value of the property is comma separated list of reference names of the supported interfaces.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <interface name="Interface1">
        <int name="Version" type="uint8" semanticType="version" />
    </interface>

    <interface name="Interface2">
        <int name="Version" type="uint8" semanticType="version" />
        <set name="Flags"> length="1">
            ...
        </set>
    </interface>


    <frame name="ProtocolFrame">
        <size name="Size">
            ...
        </size>
        <value name="Version" interfaces="Interface1, Interface2" ...>
            <int name="VersionField" type="uint8" />
        </value>
        <id name="Id">
            ...
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

In case the whole protocol schema contains no more than one <**interface**> definition, then the usage of the **interfaces** property is not necessary.

In addition to specifying the interfaces themselves, there is a need to specify the field name in the interface(s), that will hold the processed value, using **interfaceFieldName** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <interface name="Interface1">
        <int name="Version" type="uint8" semanticType="version" />
    </interface>

    <interface name="Interface2">
        <int name="Version" type="uint8" semanticType="version" />
        <set name="Flags"> length="1">
            ...
        </set>
    </interface>


    <frame name="ProtocolFrame">
        <size name="Size">
            ...
        </size>
        <value name="Version" interfaces="Interface1, Interface2">
            <interfaceFieldName value="Version" />
            <field>
                <int name="VersionField" type="uint8" />
            </field>
        </value>
        <id name="Id">
            ...
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

NOTE, that all the interfaces listed in the **interfaces** property value must have a field with the name specified as the **interfaceFieldName** value. Also the specified field's index (among other available fields of the <**interface**>) must be the same for all the listed interfaces.

Pseudo Value Layer

There are protocols that don't report protocol version in their transport framing. Instead, they use some special "connection" message that report protocol version. As the result, all subsequent messages must adhere to the reported version. The **CommsDSL** resolves this problem by defining **pseudo** <**value**> layer using **pseudo** property with boolean value. The field of the pseudo <**value**> layer does not get serialized. However, the code generator must allow external assignment of the required value to the private data members of the <**value**> layer's class.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <interface name="Interface1">
        <int name="Version" type="uint8" semanticType="version" />
    </interface>

    <message name="Connect" id="1">
        <int name="ProtocolVersion" type="uint8" />
    </message>

    <frame name="ProtocolFrame">
        <size name="Size">
            ...
        </size>
        <id name="Id">
            ...
        </id>
        <value name="Version" interfaces="Interface1" pseudo="true">
            <interfaceFieldName value="Version" />
            <field>
                <int name="VersionField" type="uint8" />
            </field>
        </value>
        <payload name="Data" />
    </frame>
</schema>
```

The location of the pseudo <**value**> layer within the <**frame**> is not important. It is recommended to define it right before the <**payload**> layer.

As part of processing **Connect** message in the example above, the client code is expected to retrieve the value of the **ProtocolVersion** field and report it to the pseudo <**value**> layer within the processing frame. For all the subsequent messaged, the pseudo <**value**> layer is responsible to assign appropriate value (version in the example above) to the newly created message object before message payload gets read. This is because such values may have an influence on how the **read** operation is executed (existence of some fields may depend on the assigned value).

Use properties table for future references.

<custom> Layer

The <**custom**> layer represent any other protocol specific layer, functionality of which cannot be impelemented using other provided layers. The code generator must allow injection of the externally implemented code of the layer functionality when generating code relevant to the transport framing.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        <size name="Size">
            ...
        </size>
        <custom name="SomeCustomLayer">
            <int name="SomeField" type="uint8" />
        </custom>
        <id name="Id">
            ...
        </id>
        <payload name="Data" />
    </frame>
</schema>
```

The <**value**> layer has all the common properties as well as extra properties and elements described below.

ID Replacement

Most protocol frames are expected to use <id> layer to process numeric ID of the message. However, there are protocols that may use the same field not only for numeric ID, but also for some extra flags. In this case <id> layer cannot be used, because it expects the whole field to be dedicated to storage of the numeric ID. In such case <**custom**> layer needs to be used but marked as one replacing the <id> with **idReplacement** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema endian="big" ...>
    <frame name="ProtocolFrame">
        <size name="Size">
            ...
        </size>
        <custom name="IdAndFlags" idReplacement="true">
            <bitfield name="IdAndFlagsField">
                <int name="Id" type="uint16" bitLength="12" />
                <set name="Flags" bitLength="4">
                    ...
                </set>
            </bitfield>
        </custom>
        <payload name="Data" />
    </frame>
</schema>
```

Use properties table for future references.

Protocol Versioning Summary

This chapter summarizes all version related aspects of the protocol definition.

Version of the Schema

The protocol definition <schema> has the **version** property, that specifies numeric version of the protocol.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" version="5">
    ...
</schema>
```

Version in the Interface

If protocol reports its version via transport framing or via some special "connection" message, and the protocol version must influence how some messages are deserialized / handled, then there is a need for <interface> definition, which must contain version field, marked as such using **semanticType="version"** property.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" version="5">
    <interface name="Interface">
        <int field="SomeName" type="uint16" semanticType="version" />
    </interface>
    ...
</schema>
```

Version in the Frame

In addition to the <interface> containing version information, the transport <frame> is also expected to contain <value> layer, which will reassign the received version information to the message object (via <interface>).

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" version="5">
    <interface name="Interface">
        <int field="SomeName" type="uint16" semanticType="version" />
    </interface>

    <frame name="Frame">
        <size name="Size">
            <int name="SizeField" type="uint16" serOffset="2"/>
        </size>
        <id name="Id">
            <int name="IdField" type="uint8" />
        </id>
        <value name="Version" interfaces="Interface" interfaceFieldName="SomeName">
            <int name="VersionField" type="uint16" />
        </value>
        <payload name="Data" />
    </frame>
</schema>
```

Even if the protocol version is communicated in one of the messages and the real framing doesn't really contain any version information, it still should be defined with similar <value> layer, but marked as **pseudo**.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="MyProtocol" version="5">
    <interface name="Interface">
        <int field="SomeName" type="uint16" semanticType="version" />
    </interface>

    <frame name="Frame">
        <size name="Size">
            <int name="SizeField" type="uint16" serOffset="2"/>
        </size>
        <id name="Id">
            <int name="IdField" type="uint8" />
        </id>
        <value name="Version" interfaces="Interface" interfaceFieldName="SomeName"
pseudo="true">
            <int name="VersionField" type="uint16" />
        </value>
        <payload name="Data" />
    </frame>
</schema>
```

The generated code is expected to allow external assignment of the protocol version information (once it's known) to inner data structures of appropriate <value> framing layer, which in turn is responsible to re-assign the relevant version information to every subsequent message object when

it's created, but before the message payload being read.

Version of the Fields and Messages

Every message and field support the following properties, wich can be used to supply version related information.

- **sinceVersion** - Version when the message / field has been introduced.
- **deprecated** - Version when the message / field has been deprecated.
- **removed** - Indication that deprecated message / field must be removed from serialization and code generation.

Version Dependency of the Code

The generated code is expected to be version dependent (i.e. presence of some messages / fields is determined by the reported version value), if **at least** one of the defined <interface> -es contains version field (marked by **semanticType="version"**).

If none of the interfaces has such field, then the generated code cannot be version dependent and all the version related properties become for documentation purposes only and cannot influence the presence of the messages / fields. In such cases the code generator is expected to receive required protocol version in its command line parameters and generate code for requested protocol version.

Compatibility Recommendation

In case **CommsDSL** is used to define new protocol developed from scratch and backward / forward compatibility of the protocol is a desired feature, then there are few simple rules below, following of which can insure such compatibility.

- Use <size> layer in the transport framing to report remaining length of the message.
- Use <value> layer to report protocol version in the transport framing or define special "connect" message that is sent to establish connection and report protocol version (mark the <value> layer to be **pseudo**).
- Always add new fields at the end of the <message> . Don't forget to specify their version using **sinceVersion** property.
- Don't **remove** deprecated fields.
- Always add new fields at the end of the <list> element.
- Add element serialization length report before every element of the <list> field (using **elemLengthPrefix** property).
- In case **elemFixedLength** property is assigned for the <list> (to avoid redundant report of the same element length before every element), never add variable length fields to the element of the list.

Appendix

This chapter contains list of properties for all the elements described earlier in this document for a quick reference.

Properties of <schema>

Refer to Schema chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	string	1	yes		Name of the protocol. Allowed to have spaces.
endian	"big" or "little"	1	no	little	Endian of the protocol.
description	string	1	no		Human readable description of the protocol.
version	unsigned	1	no	0	Version of the protocol.
dslVersion	unsigned	1	no	0	Version of the used DSL.
nonUniqueMsgIdAllowed	bool	1	no	false	Allow non-unique numeric message IDs.

Common Properties of Fields

Refer to Common Properties of Fields chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	1	yes		Name of the field.
description	string	1	no		Human readable description of the field.
reuse	reference string	1	no		Field definition of which to copy.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
displayName	string	1	no		Name of the field to display. If empty, the code generator must use value of property **name** instead. In order to force empty name to display, use "_" (underscore).
displayReadOnly	bool	1	no	false	Disable modification of the field in visual analysis tool(s).
displayHidden	bool	1	no	false	Don't display field at all in visual analysis tool(s).
sinceVersion	unsigned	1	no	0	Version of the protocol in which field was introduced. Applicable only to members of the <message> or <bundle> .
deprecated	unsigned	1	no	max unsigned	Version of the protocol in which field was deprecated. Must be greater than value of **sinceVersion**. Applipable only to members of the <message> or <bundle> .
removed	bool	1	no	false	Indicates whether deprecated field has been removed from being serialized. Applicable only to members of the <message> or <bundle> .
failOnInvalid	bool	1	no	false	Fail **read** operation if read value is invalid.
pseudo	bool	1	no	false	In case of **true**, don't serialize/deserialize this field.
customizable	bool	1	no	false	Mark the field to allow compile time customization regardless of code generator's level of customization.
semanticType	"none", "messageId", "version", "length"	1	no	none	Specify semantic type of the field. It allows code generator to generate special code for special cases. Value "length" was introduced in **v2** of this specification.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
forceGen	bool	3	no	false	Force generation of field's code regardless of it's being referenced or not.

Properties of <enum> Field

The <**enum**> field has all the common properties as well as ones listed below. Refer to <enum> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
type	"int8", "uint8", "int16", "uint16", "int32", "uint32", "int64", "uint64", "intvar", "uintvar"	1	yes		Underlying primitive type.
defaultValue	numeric or name	1	no	0	Default value. Must fit the underlying **type**.
endian	"big" or "little"	1	no	endian of schema	Endian of the field.
length	unsigned	1	no	length of **type**	Forced serialization length.
bitLength	unsigned	1	no	length of **type** in bits	Serialization length in bits, applicable only to a member of <bitfield> .
hexAssign	bool	1	no	false	Force generated code to assign enum values using hexadecimal numbers.
nonUniqueAllowed	bool	1	no	false	Allow non unique <**validValue**> -es.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
validCheckVersion	bool	1	no	false	Take into account protocol version when generating code for field's value validity check.

Properties of <validValue> Child Element of <enum> Field

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	1	yes		Name of the value.
val	numeric	1	yes		Numeric value.
description	string	1	no		Human readable description of the value.
displayName	string	1	no		Human readable name of the value to display in various analysis tools.
sinceVersion	unsigned	1	no	0	Version of the protocol in which value was introduced.
deprecated	unsigned	1	no	max unsigned	Version of the protocol in which value was deprecated. Must be greater than value of **sinceVersion**.

Properties of <int> Field

The **<int>** field has all the common properties as well as ones listed below. Refer to <int> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
type	"int8", "uint8", "int16", "uint16", "int32", "uint32", "int64", "uint64", "intvar", "uintvar"	1	yes		Underlying primitive type.
defaultValue	numeric or name	1	no	0	Default value. Must fit the underlying **type**.
endian	"big" or "little"	1	no	endian of schema	Endian of the field.
length	unsigned	1	no	length of **type**	Forced serialization length.
bitLength	unsigned	1	no	length of **type** in bits	Serialization length in bits, applicable only to a member of <bitfield> .
serOffset	numeric	1	no	0	Extra value that needs to be added to the field's value when the latter is being serialized.
signExt	bool	1	no	true	Enable / Disable sign extension of the signed value when **length** property is used to reduce the default serialization length.
scaling	"numeric / numeric"	1	no	1/1	Scaling ratio.
units	units	1	no		Units of the value.
validRange	"[numeric , numeric]"	1	no		Range of valid values.
validValue	numeric	1	no		Valid value.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
validMin	numeric	1	no		Valid minimal value. All the numbers above it are considered to be valid.
validMax	numeric	1	no		Valid maximal value. All the numbers below it are considered to be valid.
validCheckVersion	bool	1	no	false	Take into account protocol version when generating code for field's value validity check.
displayDecimals	numeric	1	no	0	Indicates to GUI analysis tools to display this field as floating point value with specified number of digits after the fraction point.
displayOffset	numeric	1	no	0	Indicates to GUI analysis tools to add specified offset value to a field's value when displaying it.
nonUniqueSpecialsAllowed	bool	2	no	false	Allow non unique <**special**>-s.
displaySpecials	bool	2	no	true	Control displaying <**special**> values in analysis tools.

Properties of <special> Child Element of <int> Field

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	1	yes		Name of the value.
val	numeric	1	yes		Numeric value.
description	string	1	no		Human readable description of the value.
sinceVersion	unsigned	1	no	0	Version of the protocol in which value was introduced.
deprecated	unsigned	1	no	max unsigned	Version of the protocol in which value was deprecated. Must be greater than value of **sinceVersion**.
displayName	string	2	no		Name to display in various analysis tools.

Properties of <set> Field

The **<set>** field has all the common properties as well as ones listed below. Refer to <set> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
type	"uint8", "uint16", "uint32", "uint64"	1	yes (no if **length** is specified)		Underlying primitive type.
length	unsigned	1	yes (no if **type** is specifed)	length of **type**	Serialization length.
bitLength	unsigned	1	no	length of **type** in bits	Serialization length in bits, applicable only to a member of <bitfield> .
defaultValue	bool	1	no	false	Default initialization value of every bit.
reservedValue	bool	1	no	false	Expected value of every reserved bit.
endian	"big" or "little"	1	no	endian of schema	Endian of the field.
nonUniqueAllowed	bool	1	no	false	Allow non unique **<bit>**-s.
validCheckVersion	bool	1	no	false	Take into account protocol version when generating code for field's value validity check.

Properties of <bit> Child Element of <set> Field

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	1	yes		Name of the value.
idx	numeric	1	yes		Index of the specified bit. Counting starts from least significant bit.
description	string	1	no		Human readable description of the bit.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
displayName	string	1	no		Human readable name of the bit to display in various analysis tools.
defaultValue	bool	1	no	**defaultValue** of the <**set**> field	Default value of the bit (when constructed).
reservedValue	bool	1	no	**reservedValue** of the <**set**> field	Expected value of the bit if it is reserved.
reserved	bool	1	no	false	Mark / Unmark the bit as being reserved.
sinceVersion	unsigned	1	no	0	Version of the protocol in which bit was introduced (became non-reserved).
deprecated	unsigned	1	no	max unsigned	Version of the protocol in which bit was deprecated (became reserved). Must be greater than value of **sinceVersion**.

Properties of <bitfield> Field

The <**bitfield**> field has all the common properties as well as ones listed below. Refer to <bitfield> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
endian	"big" or "little"	1	no	endian of schema	Endian of the field.

Extra child XML elements allowed:

XML Element	DSL Version	Description
<**members**>	1	Wraps member fields.

Properties of <bundle> Field

The <**bundle**> field has all the common properties as well as ones listed below. Refer to <bundle>

Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
reuseAliases	bool	3	no	true	Control copy of the defined aliases from reused other <bundle> when **reuse** property is used.

Extra child XML elements allowed:

XML Element	DSL Version	Description
<members>	1	Wraps member fields.
<alias>	3	Alias names for other member fields. See Aliases for more info.

Properties of <string> Field

The **<string>** field has all the common properties as well as ones listed below. Refer to <string> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
defaultValue	string	1	no		Default value.
length	unsigned	1	no	0	Fixed serialization length. **0** means no length limit. Cannot be used together with **lengthPrefix** or **zeroTermSuffix**.
lengthPrefix	field or reference	1	no		Prefix field containing length of the string. Cannot be used together with **length** or **zeroTermSuffix**.
zeroTermSuffix	bool	1	no	false	Terminate string with **0**. Cannot be used together with **length** or **lengthPrefix**.

Properties of <data> Field

The **<data>** field has all the common properties as well as ones listed below. Refer to <data> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
defaultValue	string	1	no		Default value. Case-insensitive hexadecimal string.
length	unsigned	1	no	0	Fixed serialization length. **0** means no length limit. Cannot be used together with **lengthPrefix**.
lengthPrefix	field or reference	1	no		Prefix field containing length of the data. Cannot be used together with **length**.

Properties of <list> Field

The **<list>** field has all the common properties as well as ones listed below. Refer to <list> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
element	field or reference	1	yes		Element of the field.
count	unsigned	1	no	0	Fixed number of elements in the list. Cannot be used together with **lengthPrefix** or **countPrefix**.
countPrefix	field or reference	1	no		Prefix field containing number of elements in the list. Cannot be used together with **count** or **lengthPrefix**.
lengthPrefix	field or reference	1	no		Prefix field containing serialization length of the list. Cannot be used together with **count** or **countPrefix**.
elemLengthPrefix	field	1	no		Prefix field containing serialization length of the list **element**.
elemFixedLength	bool	1	no	false	Indication of whether list has and will allways have fixed length element, so **elemLengthPrefix** prefixes only the first element and not the rest.

Properties of <float> Field

The <**float**> field has all the common properties as well as ones listed below. Refer to <float> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
type	"float", "double"	1	yes		Underlying primitive type
defaultValue	floating point value, **nan**, **inf**, **-inf**	1	no	0.0	Default value. Must fit the underlying **type**.
endian	"big" or "little"	1	no	endian of schema	Endian of the field.
units	units	1	no		Units of the value.
validRange	"[fp_value , fp_value]"	1	no		Range of valid values.
validValue	floating point value, **nan**, **inf**, **-inf**	1	no		Valid value.
validMin	floating point value	1	no		Valid minimal value. All the numbers above it are considered to be valid.
validMax	floating point value	1	no		Valid maximal value. All the numbers below it are considered to be valid.
validFullRange	bool	1	no	false	Mark all the range of existing FP values to be valid, excluding **nan**, **inf**, and **-inf**.
validCheckVersion	bool	1	no	false	Take into account protocol version when generating code for field's value validity check.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
displayDecimals	numeric	1	no	0	Indicates to GUI analysis how many digits need to be displayed after the fraction point.
nonUniqueSpecialsAllowed	bool	2	no	false	Allow non unique <**special**>-s.
displaySpecials	bool	2	no	true	Control displaying <**special**> values in analysis tools.

Properties of <special> Child Element of <float> Field

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	1	yes		Name of the value.
val	floating point value, **nan**, **inf**, **-inf**	1	yes		Numeric value.
description	string	1	no		Human readable description of the value.
sinceVersion	unsigned	1	no	0	Version of the protocol in which value was introduced.
deprecated	unsigned	1	no	max unsigned	Version of the protocol in which value was deprecated. Must be greater than value of **sinceVersion**.
displayName	string	2	no		Name to display in various analysis tools.

Properties of <ref> Field

The <**ref**> field has all the common properties as well as ones listed below. Refer to <ref> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
field	reference	1	yes		Reference to other field.
bitLength	unsigned	2	no	length in bits	Serialization length in bits, applicable only to a member of <bitfield> and when referencing a field that can be a member of <bitfield> .

Properties of <optional> Field

The <**optional**> field has all the common properties as well as ones listed below. Refer to <optional> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
field	field	1	no		Wrapped field.
defaultMode	"tentative", "missing", "exist"	1	no	tentative	Default mode of the field. See also Default Mode Strings below.
cond	string	1	no		Condition when the field exists.
displayExtModeCtrl	bool	1	no	false	Disable manual update of the mode in GUI analysis tools.

Inner field must be specified using **field** property or as child XML element.

Extra child XML elements allowed:

XML Element	DSL Version	Description
<field>	1	Wraps member field.
<cond>	1	Condition when field exists.
<and>	1	Logical "and" of conditions.
<or>	1	Logical "or" of conditions.

Default Mode Strings

Mode	Accepted Values (case insensitive)
Tentative	"tentative", "tent", "t"

Mode	Accepted Values (case insensitive)
Missing	"missing", "miss", "m"
Exist	"exist", "exists", "e"

Properties of <variant> Field

The **<variant>** field has all the common properties as well as ones listed below. Refer to <variant> Field chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
defaultMember	name or numeric index	1	no	-1	Default member. Negative number means no member selected.
displayIdxReadOnlyHidden	bool	1	no	false	Hide active index of the member when field displayed in read-only mode.

Extra child XML elements allowed:

XML Element	DSL Version	Description
<members>	1	Wraps member fields.

Units

The tables below contain lists of available units.

Units Name	Accepted Values (case insensitive)
Unknown / None	*empty string*

Time Units

Units Name	Accepted Values (case insensitive)
Nanoseconds	"ns", "nanosec", "nanosecs", "nanosecond", "nanoseconds"
Microseconds	"us", "microsec", "microsecs", "microsecond", "microseconds"
Milliseconds	"ms", "millisec", "millisecs", "millisecond", "milliseconds"
Seconds	"s", "sec", "secs", "second", "seconds"
Minutes	"min", "mins", "minute", "minutes"
Hours	"h", "hour", "hours"

Units Name	Accepted Values (case insensitive)
Days	"d", "day", "days"
Weeks	"w", "week", "weeks"

Distance Units

Units Name	Accepted Values (case insensitive)
Nanometers	"nm", "nanometer", "nanometre", "nanometers", "nanometres"
Micrometers	"um", "micrometer", "micrometre", "micrometers", "micrometres"
Millimeters	"mm", "millimeter", "millimetre, "millimeters", "millimetres"
Centimeters	"cm", "centimeter", "centimetre", "centimeters", "centimetres"
Meters	"m", "meter", "metre", "meters", "metres"
Kilometers	"km", "kilometer", "kilometre", "kilometers", "kilometres"

Speed Units

Units Name	Accepted Values (case insensitive)
Nanometers Per Second	"nm/s", "nmps", "nanometer/second", "nanometre/second", "nanometers/second", "nanometres/second"
Micrometers Per Second	"um/s", "umps", "micrometers/second", "micrometres/second", "micrometer/second", "micrometre/second"
Millimeters Per Second	"mm/s", "mmps", "millimeter/second", "millimetre/second", "millimeters/second", "millimetres/second"
Centimeters Per Second	"cm/s", "cmps", "centimeter/second", "centimetre/second", "centimeters/second", "centimetres/second"
Meters Per Second	"m/s", "mps", "meter/second", "metre/second", "meters/second", "metres/second"
Kilometers Per Second	"km/s", "kmps", "kps", "kilometer/second", "kilometre/second", "kilometers/second", "kilometres/second"
Kilometers Per Hour	"km/h", "kmph", "kph", "kilometer/hour", "kilometre/hour", "kilometers/hour", "kilometres/hour"

Frequency Units

Units Name	Accepted Values (case insensitive)
Hertz	"hz", "hertz"
Kilohertz	"khz", "kilohertz"
Megahertz	"mhz", "megahertz"
Gigahertz	"ghz", "gigahertz"

Angle Units

Units Name	Accepted Values (case insensitive)
Degrees	"deg", "degree", "degrees"
Radians	"rad", "radian", "radians"

Electrical Current Units

Units Name	Accepted Values (case insensitive)
Nanoamperes	"na", "nanoamp", "nanoamps", "nanoampere", "nanoamperes"
Microamperes	"ua", "microamp", "microamps", "microampere", "microamperes"
Milliamperes	"ma", "milliamp", "milliamps", "milliampere", "milliamperes"
Amperes	"a", "amp", "amps", "ampere", "amperes"
Kiloamperes	"ka", "kiloamp", "kiloamps", "kiloampere", "kiloamperes"

Electrical Voltage Units

Units Name	Accepted Values (case insensitive)
Nanovolts	"nv", "nanovolt", "nanovolts"
Microvolts	"uv", "microvolt", "microvolts"
Millivolts	"mv", "millivolt", "millivolts"
Volts	"v", "volt", "volts"
Kilovolts	"kv", "kilovolt", "kilovolts"

Computer Memory Units

Units Name	Accepted Values (case insensitive)
Bytes	"b", "byte", "bytes"
Kilobytes	"kb", "kilobyte", "kilobytes"
Megabytes	"mb", "megabyte", "megabytes"
Gigabytes	"gb", "gigabyte", "gigabytes"
Terabytes	"tb", "terabyte", "terabytes"

Properties of <message>

Refer to Messages chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	1	yes		Name of the message.
id	numeric	1	yes		Numeric ID of the message.
description	string	1	no		Human readable description of the message.
displayName	string	1	no		Name of the message to display. If empty, the code generator must use value of property **name** instead.
copyFieldsFrom	reference string	1	no		Message definition from which fields need to be copied.
order	numeric	1	no	0	Relative order of the messages with the same **id**.
sinceVersion	unsigned	1	no	0	Version of the protocol in which message was introduced.
deprecated	unsigned	1	no	max unsigned	Version of the protocol in which message was deprecated. Must be greater than value of **sinceVersion**.
removed	bool	1	no	false	Indicates whether deprecated message has been removed from being supported.
sender	"both", "client", "server"	1	no	both	Endpoint that sends the message.
customizable	bool	1	no	false	Mark the message to allow compile time customization regardless of code generator's level of customization.
copyFieldsAliases	bool	3	no	true	Control copy of the defined aliases when **copyFieldsFrom** property is used to copy fields from the other <message> .

Extra child XML elements allowed:

XML Element	DSL Version	Description
<fields>	1	Wraps member fields.

XML Element	DSL Version	Description
<alias>	3	Alias names for other member fields. See Aliases for more info.

Properties of <interface>

Refer to Interfaces chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	1	yes		Name of the interface.
description	string	1	no		Human readable description of the interface.
copyFieldsFrom	reference string	1	no		Interface definition from which fields need to be copied.
copyFieldsAliases	bool	3	no	true	Control copy of the defined aliases when **copyFieldsFrom** property is used to copy fields from the other <interface> .

Extra child XML elements allowed:

XML Element	DSL Version	Description
<fields>	1	Wraps member fields.
<alias>	3	Alias names for other member fields. See Aliases for more info.

Properties of <alias>

Refer to Aliases chapter for detailed description. Introduced in DSL version **3**.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	3	yes		Name of the alias.
description	string	3	no		Human readable description of the alias.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
field	relative reference string	3	yes		Reference to the aliased field, must start with $ character.

Properties of <frame>

Refer to Frames chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	1	yes		Name of the frame.
description	string	1	no		Human readable description of the frame.

Extra child XML elements allowed:

XML Element	DSL Version	Description
<layers>	1	Wraps member layers.

Common Properties of Frame Layers

Refer to Common Properties of Layers chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
name	name string	1	yes		Name of the layer.
description	string	1	no		Human readable description of the layer.
field	field or reference to it	1	yes (except for <payload>)		Wrapped field definition.

Properties of <checksum> Frame Layer

The <**checksum**> layer has all the common properties as well as ones listed below. Refer to <checksum> Layer chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
alg	"sum", "crc-ccitt", "crc-16", "crc-32", "custom"	1	yes		Checksum calculation algorithm.
algName	name string	1	no (unless **alg** is **custom**)		Name of the custom algorithm. Applicable only if **alg="custom"**.
from	name string	1	yes (only if **until** is not specified)		Name of the frame layer, from which the checksum calculation starts.
until	name string	1	yes (only if **from** is not specified)		Name of the frame layer, until (and including) which the checksum calculation is executed.
verifyBeforeRead	bool	1	no	false	Perform checksum verification without reading values of other layers.

Properties of <value> Frame Layer

The <**value**> layer has all the common properties as well as ones listed below. Refer to <value> Layer chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
interfaces	comma separated list of names	1	no		List of supported <interface> -es.
interfaceFieldName	name string	1	yes (only if **interfaces** is not empty)		Name of the relevant field inside each <interface> .
pseudo	bool	1	no	false	Mark the layer as **pseudo** one, i.e. one that doesn't serialize its field.

Properties of <custom> Frame Layer

The <**custom**> layer has all the common properties as well as ones listed below. Refer to <custom> Layer chapter for detailed description.

Property Name	Allowed Type / Value	DSL Version	Required	Default Value	Description
idReplacement	bool	1	no	false	Mark the layer as one replacing <id> .