CommsDSL Specification

Contents

1 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.

1.1 Specification Version

Current version of this document is **3.1**

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.

1.2 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>
```

1.2.1 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.

1.2.2 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>
```

1.2.3 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>
```

1.2.4 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.

1.3 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>
```

1.4 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](schema_def.md) 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>
```

1.5 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>
```

1.6 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>
```

1.7 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.

1.8 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>
```

1.9 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" />
```

1.10 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>
```

1.11 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.

2 Schema

Schema definition may contain various global (protocol-wide) [properties](../intro/properties.md).

## 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](../fields/fields.md) 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](../intro/properties.md) 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](../intro/protocol_versioning.md) 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](https://semver.org/) 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](../messages/messages.md) 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](../appendix/schema.md) for future references.

## 2.1 Fields

Any **field** can be defined as independent element inside the **<fields>** child of the [<schema>](../intro/schema_def.md) or a [namespace](../intro/namespaces.md). ``` <?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>](../intro/schema_def.md) or [<ns>](../intro/namespaces.md) 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>](enum.md) - Enumeration field. - [<int>](int.md) - Integral value field. - [<set>](set.md) - Bitset (bitmask) field. - [<bitfield>](bitfield.md) - Bitfield field. - [<bundle>](bundle.md) - Bundle field. - [<string>](string.md) - String field. - [<data>](data.md) - Raw data field. - [<list>](list.md) - List of other fields. - [<float>](float.md) - Floating point value field. - [<ref>](ref.md) - Reference to (alias of) other field. - [<optional>](optional.md)- Optional field. - [<variant>](variant.md)- Variant field.

All this fields have [common](common.md) as well as their own specific set of properties.

# 3 Messages

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

## Message Name Every message definition must specify its [name](../intro/names.md) using **name** [property](../intro/properties.md). ``` <?xml version="1.0" encoding="UTF-8"?> <schema …> <message name="Msg1" …> … </message> </schema> ```

## Numeric ID Every message definition must specify its [numeric](../intro/numeric.md) ID using **id** [property](../intro/properties.md). ``` <?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>](../fields/enum.md) 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](../intro/properties.md) 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](../intro/properties.md)) rather than using a [name](../intro/names.md). ``` <?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](../intro/properties.md) 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](../fields/fields.md) 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](../intro/properties.md) defined as XML child
of the **&lt;message&gt;**, then all the fields must be wrapped in
**&lt;fields&gt;** 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](#numeric-id), it is required to specify order in which they are expected to be processed (read). The ordering is specified using **order** [property](../intro/properties.md) with unsigned [numeric](../intro/numeric.md) 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](../schema/schema.md) 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](../schema/schema.md). 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>](../intro/platforms.md) XML nodes. Every message may list platforms in which it must be supported using **platforms** [property](../intro/properties.md). 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](../intro/boolean.md) 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](../aliases/aliases.md) chapter for more details.

Use [properties table](../appendix/message.md) for future references.

4 Interfaces

There are protocols that attach some extra information, such as version and/or extra flags, to the message transport [framing](../frames/frames.md). 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](../fields/fields.md). ``` <?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](../frames/frames.md).

## Interface Name Every interface definition must specify its [name](../intro/names.md) using **name** [property](../intro/properties.md). ``` <?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](../intro/properties.md) 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](../fields/fields.md) that can be specified as child XML elements. If there is any other [property](../intro/properties.md) 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](../versioning/versioning.md). 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 an 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](../aliases/aliases.md) chapter for more details.

Use [properties table](../appendix/interface.md) for future references.

5 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](../intro/names.md) using **name** [property](../intro/properties.md). ``` <?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](../intro/properties.md) 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](../fields/fields.md), 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>](payload.md) - Message payload. - [<id>](id.md) - Numeric message ID. - [<size>](size.md) - Remaining size (length). - [<sync>](sync.md) - Synchronization bytes. - [<checksum>](checksum.md) - Checksum. - [<value>](value.md) - Extra value, usually to be assigned to one of the [<interface>](../interfaces/interfaces.md) fields. - [<custom>](custom.md) - Any other custom layer, not defined by **CommsDSL**.

If there is any other [property](../intro/properties.md) 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="Protocol /> <layers> <id name="Id" field="MsgId" /> <payload name="Data" /> </layers> </frame> </schema> ```

All these layers have [common](common.md) as well as their own specific set of properties.

Use [properties table](../appendix/frame.md) 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>](../schema/schema.md) has the **version** [property](../intro/properties.md), 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](../frames/frames.md) or via some special "connection" [message](../messages/messages.md), and the protocol version must influence how some messages are deserialized / handled, then there is a need for [<interface>](../interfaces/interfaces.md) 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>](../interfaces/interfaces.md) containing version information, the transport [<frame>](../frames/frames.md) is also expected to contain [<value>](../frames/value.md) layer, which will reassign the received version information to the message object (via [<interface>](../interfaces/interfaces.md)). ``` <?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>
```
~~~~

## Version of the Fields and Messages Every [message](../messages/messages.md) and [field](../fields/fields.md) 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>](../interfaces/interfaces.md)-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>](../frames/size.md) layer in the transport [framing](../frames/frames.md) to report remaining length of the message. - Use [<value>](../frames/value.md) layer to report protocol version in the transport [framing](../frames/frames.md) or define special "connect" message that is sent to establish connection and report protocol version (mark the [<value>](../frames/value.md) layer to be **pseudo**). - Always add new [fields](../fields/fields.md) at the end of the [<message>](../messages/messages.md). Don't forget to specify their version using **sinceVersion** property. - Don't **remove** deprecated [fields](../fields/fields.md). - Always add new [fields](../fields/fields.md) at the bottom of the [<list>](../fields/list.md) element. - Add element serialization length report before every element of the [<list>](../fields/list.md) field (done using **elemLengthPrefix** property). - In case **elemFixedLength** [property](../intro/properties.md) is assigned for the [<list>](../fields/list.md) (to avoid redundant report of the same element length before every element), never add variable length [fields](../fields/fields.md) to the element of the list.