An OMG® DDS® Consolidated JSON Syntax Publication

DDS Consolidated JSON Syntax

Version 1.0

OMG Document Number	formal/2020-12-01
Normative Reference:	https://www.omg.org/spec/DDS-JSON
Release Date:	March 2021

Normative Machine Consumable Files:

https://www.omg.org/spec/DDS-JSON/20190601/dds-json_types.schema.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_qos.schema.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_domains.schema.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_domainparticipants.schema.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_applications.schema.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_data_samples.schema.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_dds_system.schema.json

Informative Machine Consumable Files:

https://www.omg.org/spec/DDS-JSON/20190601/dds-json_types_example.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_qos_example.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_domains_example.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_domainparticipants_example.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_applications_example.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_data_samples_example.json
https://www.omg.org/spec/DDS-JSON/20190601/dds-json_dds_system_example.json

---

USE OF SPECIFICATION – TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 9C Medway Road, PMB 274, Milford, MA 01757, U.S.A.

TRADEMARKS

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: https://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page https://www.omg.org, under Documents, Report a Bug/Issue.

Table of Contents

Table of Figures

Table of Tables

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language®); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel™); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at https://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

https://www.omg.org/spec

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
9C Medway Road, PMB 274
Medway, MA 01757
USA

Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification by completing the Issue Reporting Form listed on the main web page https://www.omg.org, under Documents, Report a Bug/Issue.

1 Scope

JavaScript Object Notation (JSON) is a lightweight language-independent text format to represent structured data. Originally inspired by the object literals of JavaScript, JSON has become an extremely popular mechanism for data interchange; information storage, with built-in support in many database management systems; and structured document definition.

This specification defines a consolidated JSON syntax to represent DDS resources and data. That is, syntax to represent the DDS Type System, DDS QoS Policies, DDS Entities and Applications, and DDS Data Samples using JSON. The syntax defined in this specification can be used as an alternative to the existing XML syntax to represent DDS resources and data defined in [DDS-XML].

2 Conformance Criteria

This document contains no independent conformance points. Rather, it defines JSON schema files [JSON-SCHEMA] to describe DDS resources that can be referenced by other specifications, leaving the definition of conformance criteria to the referencing specifications. Nevertheless, the general organization of the clauses (by means of atomic building blocks and building block sets that group them) is intended to ease conformance description and scoping.

Users of this standard shall follow these rules:

1. Future specifications that describe DDS resources in JSON shall reference this specification or a future revision thereof.
2. Future revisions of current specifications that describe DDS resources in JSON should reference this specification or a future revision thereof. Reference to this specification shall result in a selection of building blocks where all selected building blocks shall be supported entirely.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

[DDS] OMG, Data Distribution Service, Version 1.4, https://www.omg.org/spec/DDS

[DDS-XML] OMG, DDS Consolidated XML Syntax, Version 1.0, https://www.omg.org/spec/DDS-XML

[DDS-XTYPES] OMG, Extensible And Dynamic Topic Types For DDS, Version 1.3, https://www.omg.org/spec/DDS-XTypes

[ECMA-404] Ecma International, The JSON Data Interchange Syntax, https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[JSON-SCHEMA] A. Wright, H. Andrews, JSON Schema: A Media Type for Describing JSON Documents, https://tools.ietf.org/html/draft-handrews-json-schema-01

[RFC-4648] IETF, The Base16, Base32, and Base64 Data Encodings, https://tools.ietf.org/html/rfc4648

[RFC-7493] IETF, The I-JSON Message Format, https://tools.ietf.org/html/rfc7493

[RFC-8259] IETF, The JavaScript Object Notation (JSON) Data Interchange Format, https://tools.ietf.org/html/rfc8259

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Building Block

A *building block* is a consistent set of JSON schemas that together can be used to describe the syntax of JSON documents that represent a set of set of DDS resources or data. Building blocks are atomic, which means that if selected they must be totally supported.

Building blocks are described in Chapter 7, JSON Syntax for DDS Resources.

Building Block Set

A *building block set* is a selection of building blocks that determines a specific JSON schema usage.

Building block sets are described in Chapter 8, Building Block Sets.

5 Symbols

The acronyms used in this specification are show in Table 5.1.

Table 5.1: Acronyms

Acronym	Meaning
DDS	Data Distribution Service
JSON	JavaScript Object Notation
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QoS	Quality of Service
XML	Extensible Markup Language
XTypes	eXtensible and dynamic topic Types (for DDS)

6 Additional Information

6.1 Changes to Adopted OMG Specifications

This specification does not change any adopted OMG specification.

6.2 Acknowledgments

The following companies submitted this specification:

- Real-Time Innovations, Inc.

The following companies supported this specification:

- ADLINK Technology Ltd.
- Kongsberg Defence & Aerospace
- Jackrabbit Consulting
- MITRE
- Object Computing, Inc.
- Twin Oaks Computing, Inc.

This page intentionally left blank.

7 JSON Syntax for DDS Resources

7.1 JSON Representation Syntax

7.1.1 General Rules

The JSON representation of DDS-related resources shall follow these syntax rules:

- It shall be a well-formed JSON document according to the grammar rules defined in Clause 2 of [RFC-8259] and the conformance rules defined in Clause 2 of [ECMA-404].
- It shall be compliant with the I-JSON profile defined in [RFC-7493].

7.1.2 JSON Schema Definition Files

This specification makes use of the JSON Schema vocabulary specified in [JSON-SCHEMA] to represent the syntax of the different building blocks that define DDS resources. In particular, each building block provides a normative JSON schema file that defines its syntax (see Clause 7.3.1).

7.2 JSON Representation of Resources Defined in the DDS IDL PSM

The JSON representation of resources that correspond to data types defined in the DDS IDL PSM [DDS] is obtained by performing a one-to-one mapping of the corresponding IDL type according to the rules specified in this clause.

7.2.1 JSON Representation of Enumeration Types

IDL Enumerations are represented in JSON using string types that may only be assigned the string representation of the corresponding enumeration literals.

7.2.1.1 Example (Non-normative)

For example, **`HistoryQosPolicyKind`** is defined in the DDS IDL PSM as:

```
enum HistoryQosPolicyKind {
    KEEP_LAST_HISTORY_QOS,
    KEEP_ALL_HISTORY_QOS
};
```

The equivalent representation in JSON is defined by the JSON schema below:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "definitions": {
        ...
        "HistoryQosPolicyKind": {
            "enum": [
                "KEEP_LAST_HISTORY_QOS",
                "KEEP_ALL_HISTORY_QOS"
            ],
            "type": "string",
            "default": "KEEP_LAST_HISTORY_QOS"
        },
        ...
        "properties": {
            "kind": {
                "$ref": "#/definitions/HistoryQosPolicyKind"
            }
        }
        ...
    },
    ...
```

```
}
```

An example JSON resource representation satisfying this syntax would be:

```
{
    "kind": "KEEP_ALL_HISTORY_QOS"
}
```

Conversely, the following JSON resource representation would not satisfy the syntax:

```
{
    "kind": "A_STRING_VALUE"
}
```

7.2.2 JSON Representation of Primitive Constants

The DDS IDL PSM defines constant values of type **`long`** and **`string`**. These are intended as predefined values that can be used to initialize members of certain structured types.

Constant definitions appear in JSON schemas as **`integer`** or **`string`** types that provide custom syntax allowing an element to have a value that is either given as a number or as a string with the constant name.

7.2.2.1 Example (Non-Normative)

For example, the DDS IDL PSM defines the constants:

```
const long LENGTH_UNLIMITED = -1;
const long DURATION_INFINITE_SEC   = 0x7fffffff;
const unsigned long DURATION_INFINITE_NSEC  = 0x7fffffff;
const long DURATION_ZERO_SEC = 0;
const unsigned long DURATION_ZERO_NSEC = 0;
const long TIME_INVALID_SEC = -1;
const unsigned long TIME_INVALID_NSEC = 0xffffffff;
```

The constant **`LENGTH_UNLIMITED`** is intended to initialize structure members that represent lengths. Constants with **`DURATION_`** prefix are intended to initialize members of the **`Duration_t`** structure and constants with **`TIME_`** prefix are intended to initialize members of the **`Time_t`** structure.

For example, the above constants are mapped into the following definitions in JSON schema format:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    ...
    "definitions": {
        ...
        "nonNegativeInteger_Duration_SEC": {
            "type": [
                "integer",
                "string"
            ],
            "pattern": "DURATION_INFINITY|DURATION_INFINITE_SEC",
            "minimum": 0,
            "examples": [
                0,
                1,
                "DURATION_INFINITY",
                "DURATION_INFINITE_SEC"
            ]
        },
        "nonNegativeInteger_Duration_NSEC": {
            "type": [
                "integer",
                "string"
            ],
            "pattern": "DURATION_INFINITY|DURATION_INFINITE_NSEC",
```

```
            "minimum": 0,
            "examples": [
                0,
                1,
                "DURATION_INFINITY",
                "DURATION_INFINITE_NSEC"
            ]
        },
        "positiveInteger_UNLIMITED": {
            "type": [
                "integer",
                "string"
            ],
            "pattern": "LENGTH_UNLIMITED",
            "minimum": 1,
            "examples": [
                1,
                2,
                "LENGTH_UNLIMITED"
            ]
        },
        ...
    },
    ...
}
```

See Clause 7.2.5 for a description on how these definitions are used to represent **`Duration_t`**.

7.2.3 JSON Representation of Structure Types

In general, IDL structures are represented in JSON as object types. The members of the IDL structure become unordered properties of the object with the member name appearing as the property name. The mapping is applied recursively for nested structures.

If the DDS specification defines default values for the structure members, the corresponding JSON element shall provide the same default value.

7.2.3.1 Example (Non-normative)

For example, **`HistoryQosPolicy`** is defined in the DDS IDL PSM [DDS] as:

```
struct HistoryQosPolicy {
    HistoryQosPolicyKind kind;
    long depth;
};
```

The DDS IDL PSM states that the default value for the **`HistoryQosPolicy`** is **`KEEP_LAST_HISTORY_QOS`** and the default depth is 1.

The equivalent representation in JSON schema is defined below:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "definitions": {
        ...
        "HistoryQosPolicy": {
            "type": "object",
            "properties": {
                "kind": {
                    "$ref": "#/definitions/HistoryQosPolicyKind"
                },
                "depth": {
                    "type": "integer",
```

```
                    "minimum": 1,
                    "default": 1
                }
            }
        }
    },
    ...
}
```

An example JSON representation satisfying this syntax would be:

```
{
    "kind": "KEEP_LAST_HISTORY_QOS",
    "depth": 10
}
```

7.2.4 JSON Representation of Arrays and Sequences

In general, IDL arrays and sequences shall be represented as JSON arrays. Nested inside each item shall be the JSON schema obtained from mapping the IDL type of the element itself to JSON.

7.2.4.1 Example (Non-normative)

For example, `QosPolicyCountSeq` is defined in the DDS IDL PSM as:

```
struct QosPolicyCount {
    long policy_id;
    long count;
};
typedef sequence<QosPolicyCount> QosPolicyCountSeq;
```

The equivalent representation in JSON is defined by the JSON schema defined below:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "definitions": {
        ...
        "QosPolicyCount": {
            "type": "object",
            "properties": {
                "policy_id": {
                    "type": "integer",
                    "minimum": 0
                },
                "count": {
                    "type": "integer",
                    "minimum": 0
                }
            }
        },
        "QosPolicyCountSeq": {
            "type": "array",
            "items": {
                "$ref": "#/definitions/QosPolicyCount"
            }
        }
    },
    ...
}
```

An example JSON representation satisfying this syntax would be:

```
[
    {
        "policy_id": 1,
```

```
        "count": 23
    },
    {
        "policy_id": 4,
        "count": 44
    }
]
```

7.2.5 JSON Representation of Duration

The IDL structure **`Duration_t`** shall be represented in JSON following the general rules for structures defined in Clause 7.2.3, except that the schema shall provide also an option to represent infinite duration—based on the constants defined for that purpose in the DDS IDL PSM.

The **`Duration_t`** structure is defined in the DDS IDL PSM as:

```
struct Duration_t {
    long sec;
    unsigned long nanosec;
};
```

The equivalent representation in JSON is defined by the following JSON schema:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "definitions": {
        "nonNegativeInteger_Duration_SEC": {
            "type": [
                "integer",
                "string"
            ],
            "pattern": "DURATION_INFINITY|DURATION_INFINITE_SEC",
            "minimum": 0,
            "examples": [
                0,
                1,
                "DURATION_INFINITY",
                "DURATION_INFINITE_SEC"
            ]
        },
        "nonNegativeInteger_Duration_NSEC": {
            "type": [
                "integer",
                "string"
            ],
            "pattern": "DURATION_INFINITY|DURATION_INFINITE_NSEC",
            "minimum": 0,
            "examples": [
                0,
                1,
                "DURATION_INFINITY",
                "DURATION_INFINITE_NSEC"
            ]
        },
        "duration": {
            "type": "object",
            "properties": {
                "sec": {
                    "$ref": "#/definitions/nonNegativeInteger_Duration_SEC"
                },
                "nanosec": {
                    "$ref": "#/definitions/nonNegativeInteger_Duration_NSEC"
                }
            }
```

```
        },
        ...
    },
    ...
}
```

7.2.5.1 Example (Non-normative)

An example JSON resource representation satisfying the syntax defined above would be:

```
{
    "duration": {
        "sec": 0,
        "nanosec": "DURATION_INFINITY_NSEC"
    }
}
```

7.3 Building Blocks

7.3.1 Overview

This specification breaks the syntax to represent DDS resources in JSON into the six different building blocks as shown in Figure 7.1:

- Building Block QoS
- Building Block Types
- Building Block Domains
- Building Block DomainParticipants
- Building Block Applications
- Building Block Data Samples

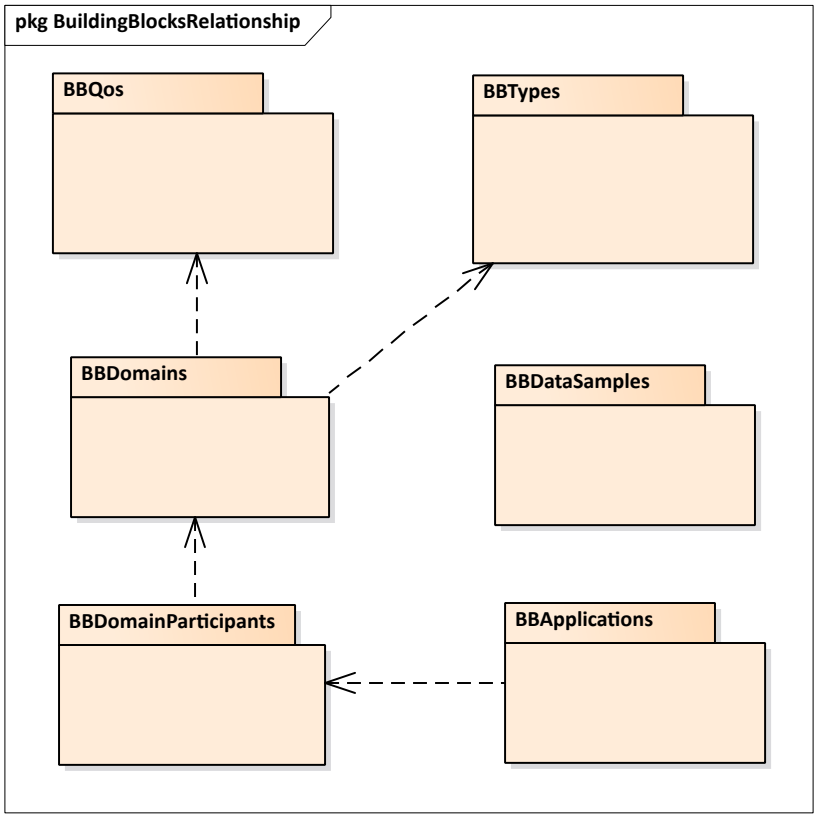

Figure 7.1: Relationship between building blocks

Each of these building blocks is associated with a normative JSON schema file:

- *dds-json_<building_block_name>.schema.json* contains the type declarations for all the constructs the building block defines. This JSON schema file may be easily integrated into other JSON schemas to define custom elements, making use of constructs from different building blocks without any restriction in terms of object hierarchy.

Moreover, each building block is associated with a non-normative JSON example file:

- *dds-json_<building_block_name>_example.json* contains an example JSON file that shows the definitions of the normative schema in practice.

7.3.2 Building Block QoS

7.3.2.1 Purpose

This building block defines the syntax to represent DDS QoS policies in JSON.

7.3.2.2 Dependencies with other Building Blocks

This building block has no dependencies on other building blocks.

7.3.2.3 Syntax

The following normative JSON schema file defines the syntax to represent DDS QoS policies in JSON format:

- *dds-json_qos.schema.json*

Moreover, the following non-normative file contains an example on how to apply the aforementioned schema to represent QoS policies in JSON format:

- *dds-json_qos_example.json*

7.3.2.4 Explanations and Semantics

7.3.2.4.1 QoS Libraries and QoS Profiles

QoS Libraries are the top level element of the Building Block QoS. They are collections of QoS Profiles, which group a set of related QoS Policies.

7.3.2.4.1.1 Example (Non-normative)

```
{
    "name": "ReliableProfilesLibrary",
    "qos_profiles": [
        {
            "name": "StrictReliableCommunicationProfile",
            "datawriter_qos": {
                "history": {
                    "kind": "KEEP_ALL_HISTORY_QOS"
                },
                "reliability": {
                    "kind": "RELIABLE_RELIABILITY_QOS"
                }
            },
            "datareader_qos": {
                "history": {
                    "kind": "KEEP_ALL_HISTORY_QOS"
                },
                "reliability": {
                    "kind": "RELIABLE_RELIABILITY_QOS"
                }
            }
        }
    ]
}
```

7.3.2.4.2 QoS Profile Inheritance

A QoS Profile can inherit from another QoS Profile using the **`"base_name"`** property. The name of the base profile shall be preceded by the name of the containing QoS Library and two separating colons (i.e., "`::`"), according to the following expression: "**`<baseQosProfileLibraryName>::<baseQosProfileName>`**".

7.3.2.4.2.1 Example (Non-normative)

```
{
    "name": "MyQosProfile",
    "base_name": "BaseQosProfileLibraryName::BaseQosProfileName",
    ...
}
```

7.3.2.4.3 QoS Profile Topic-name Filters

A QoS Profile may contain several DataWriter, DataReader, and Topic QoS settings that are selected based on the evaluation of a filter expression on the topic name. In that case, the **`"datawriter_qos"`**, **`"datareader_qos"`**, and **`"topic_qos"`** properties shall be represented as a JSON array of objects describing the entity QoS, with a **`"name"`** property and a **`"topic_filter"`** property.

The filter expression is specified via the **"topic_filter"** property in the definition of the entity QoS. If the topic filter is unspecified, the filter "*" will be assumed. The QoS with an explicit **"topic_filter"** property definition will be evaluated in order; they take precedence over a QoS without a topic filter expression.

7.3.2.4.3.1 Example (Non-normative)

For example, in the following definition:

```
{
    "name": "MyQosProfile",
    "datawriter_qos": [
        {
            "name": "DataWriterQosA",
            "topic_filter": "A*",
            "history": {
                "kind": "KEEP_ALL_HISTORY_QOS"
            },
            "reliability": {
                "kind": "RELIABLE_RELIABILITY_QOS"
            }
        },
        {
            "name": "DataWriterQosB"
            "topic_filter": "B*",
            "history": {
                "kind": "KEEP_ALL_HISTORY_QOS"
            },
            "reliability": {
                "kind": "BEST_EFFORT_RELIABILITY_QOS"
            },
            "resource_limits": {
                "max_samples": 128,
                "max_samples_per_instance": 128,
                "initial_samples": 128,
                "max_instances": 1,
                "initial_instances": 1
            }
        },
        ...
    ],
    ...
}
```

DataWriters of Topics with names matching the **"A*"** expression will have their **DataWriterQos** policies defined in the object containing the **"topic_filter": "A*"** property (i.e., **"DataWriterQosA"**). DataWriters of Topics with names matching the **"B*"** expression will have their **DataWriterQos** policies defined in the object containing the **"topic_filter": "B*"** property (i.e., **"DataWriterQosB"**).

7.3.2.4.4 QoS Profiles with a Single QoS

The definition of QoS Policies for DDS Entities within a QoS Library is a shortcut for defining a QoS Profile with QoS settings for a single DDS Entity.

7.3.2.4.4.1 Example (Non-normative)

For example, the following definition:

```
{
    "name": "MyQosLibrary",
    "datawriter_qos": {
        "name": "KeepAllWriter",
        "history": {
            "kind": "KEEP_ALL_HISTORY_QOS"
        }
```

```
    }
}
```

Is equivalent to the following:

```
{
    "name": "MyQosLibrary",
    "qos_profiles": [
        {
            "name": "KeepAllWriterProfile",
            "datawriter_qos": {
                "history": {
                    "kind": "KEEP_ALL_HISTORY_QOS"
                }
            }
        }
    ]
}
```

7.3.3 Building Block Types

7.3.3.1 Purpose

This building block gathers the syntax used to represent DDS Types in JSON. Additionally, it provides capabilities that are necessary or convenient for the organization and management of types and other JSON resource representations.

7.3.3.2 Dependencies with other Building Blocks

This building block has no dependencies on other building blocks.

7.3.3.3 Syntax

The following normative JSON schema file defines the syntax to represent all the types defined in the DDS type system in JSON format:

- *dds-json_types.schema.json*

Moreover, the following non-normative file contains an example on how to apply the aforementioned schema to represent DDS types in JSON format:

- *dds-json_types_example.json*

7.3.4 Building Block Domains

7.3.4.1 Purpose

This building block defines the syntax used to represent DDS Domains in JSON. Domains provide a data space where information can be shared by reading and writing a set of Topics, which are associated to registered data types.

7.3.4.2 Dependencies with other Building Blocks

This building block depends on Building Block QoS and Building Block Types.

7.3.4.3 Syntax

The following normative JSON schema file defines the syntax to represent DDS Domains in JSON format:

- *dds-json_domains.schema.json*

Moreover, the following non-normative file contains an example on how to apply the aformentioned schema to represent DDS Domains in JSON format:

- *dds-json_domains_example.json*

7.3.4.4 Explanations and Semantics

7.3.4.4.1 Defining a Domain

A Domain includes a set of Topics and Registered Types that can be read and written within the Domain.

Registered types shall provide a reference to data types that have been previously defined using the **"type_ref"** property of the object representing the registered type. The name under which types are registered may be different than original type name.

Topics shall refer to a registered type using the **"register_type_ref"** property of the object representing the Topic. Topics may also specify QoS settings inline following the syntax defined in the Building Block QoS. The syntax supports QoS Profile inheritance through the **"base_name"** property, as specified in Clause 7.3.2.4.2.

7.3.4.4.1.1 Example (Non-normative)

```
{
    "name": "MyDomain",
    "domain_id": 10,
    "register_types": [
        {
            "name": "MyFirstRegisterType",
            "type_ref": "MyType"
        },
        {
            "name": "MySecondRegisterType",
            "type_ref": "MyType"
        }
    ],
    "topics": [
        {
            "name": "FirstTopic",
            "register_type_ref": "MyFirstRegisterType",
            "topic_qos": {
                "base_name": "BaseQoSProfile"
            }
        },
        {
            "name": "SecondTopic",
            "register_type_ref": "MySecondRegisterType"
        }
    ]
}
```

7.3.4.4.2 Domain Inheritance

A Domain can inherit from another Domain using the **"base_name"** property of the JSON object representing the Domain. The base domain name shall be preceded by the name of the containing Domain Library and two separating colons (i.e., **"::"**), according to the following expression: **"<baseDomainLibraryName>::<baseDomainName>"**.

7.3.4.4.2.1 Example (Non-normative)

```
{
    "name": "MyDomain",
    "base_name": "BaseDomainLibraryName::BaseDomain",
    ...
}
```

7.3.5 Building Block DomainParticipants

7.3.5.1 Purpose

This building block defines the syntax to represent DDS DomainParticipants and all their contained entities (i.e., Publishers, Subscribers, DataWriters, and DataReaders) in JSON.

7.3.5.2 Dependencies with other Building Blocks

This building block depends on Building Block QoS, Building Block Types, and Building Block Domains.

7.3.5.3 Syntax

The following normative JSON schema file defines the syntax to represent DDS entities in JSON format:

- *dds-json_domainparticipants.schema.json*

Moreover, the following non-normative file contains an example on how to apply the aforementioned schema to represent DDS entities in JSON format:

- *dds-json_domainparticipants_example.json*

7.3.5.4 Explanations and Semantics

7.3.5.4.1 DomainParticipant Libraries, DomainParticipants, and Contained Entities

DomainParticipant Libraries are collections of DomainParticipants and contained entities. They are the top level elements of the Building Block DomainParticipants.

DomainParticipants are responsible for the creation and deletion of Publishers and Subscribers, which are in turn responsible for the creation and deletion of DataWriters and DataReaders.

To represent this hierarchical relationship between DDS entities, each entity is declared as a nested JSON property within the declaration of its parent entity.

7.3.5.4.1.1 Example (Non-normative)

```
{
    "name": "MyDomainParticipantLibrary",
    "domain_participants": [
        {
            "name": "MyDomainParticipant",
            "domain_ref": "MyDomainLibrary::MyDomain",
            "publishers": [
                {
                    "name": "MyPublisher",
                    "data_writers": [
                        {
                            "name": "MyDataWriter",
                            "topic_ref": "MyTopic"
                        }
                    ]
                }
            ],
            "subscribers": [
                {
                    "name": "MySubscriber",
                    "data_readers": [
                        {
                            "name": "MyDataReader",
                            "topic_ref": "MyTopic"
                        }
```

```
                    ]
                }
            ]
        }
    ]
}
```

7.3.5.4.2 Using the Domain Building Block

DomainParticipants may refer to a Domain declared in the context of a Domain Library (see Building Block Domains) using the **"domain_ref"** property of the corresponding JSON object. This makes the Topics and Registered Types defined in the Domain available for all the DataWriters and DataReaders defined in the context of the DomainParticipant.

The Domain ID specified in the parent Domain can be overridden via the **"domain_id"** property of the DomainParticipant's JSON object.

7.3.5.4.2.1 Example (Non-normative)

```
{
    "name": "MyDomainParticipant",
    "domain_ref": "MyDomainLibrary::MyDomain",
    "domain_id": 32,
    ...
}
```

7.3.5.4.3 DomainParticipant Inheritance

A DomainParticipant may inherit from another DomainParticipant defined in the context of a DomainParticipant Library using the **"base_name"** property of the corresponding JSON object. The name of the base DomainParticipant shall be preceded by the name of the containing DomainParticipant Library and two separating colons (i.e., **"::"**), according to the following expression:
"<baseDomainParticipantLibraryName>::<baseDomainParticipantName>".

7.3.5.4.3.1 Example (Non-normative)

```
{
    "name": "MyDomainParticipantLibrary",
    "domain_participants": [
        {
            "name": "MyDomainParticipant",
            "base_name": "BaseDomainParticipantLibraryName::BaseDomainParticipantName",
            ...
        }
    ]
}
```

7.3.5.4.4 Inline Entity QoS Settings Definition

Inline definition of QoS Policies is allowed in the context of an entity definition. Inline QoS settings apply only to the entity that is being defined. These definitions support QoS Profile inheritance through the **"base_name"** property as specified in Clause 7.3.2.4.2.

7.3.5.4.4.1 Example (Non-normative)

```
{
    "name": "MyDomainParticipantLibrary",
    "domain_participants": [
        {
            "name": "MyDomainParticipant",
            ...
            "domain_participant_qos": {
                "base_name": "BaseQosLibraryName::BaseQosProfileName",
```

```
                "entity_factory": {
                    "autoenable_created_entities": false
                }
            },
            ...
        }
    ]
}
```

7.3.6 Building Block Applications

7.3.6.1 Purpose

This building block defines syntax to represent DDS applications that participate (or may be participating) in the DDS Global Data Space in JSON format.

7.3.6.2 Dependencies with other Building Blocks

This building block depends on Building Block QoS, Building Block Types, Building Block Domains, and Building Block DomainParticipants.

7.3.6.3 Syntax

The following normative JSON schema file defines the syntax to represent DDS applications and their contained entities in JSON format:

- *dds-json_applications.schema.json*

Moreover, the following non-normative file contains an example on how to apply the aforementioned schema to represent applications in JSON format:

- *dds-json_applications_example.json*

7.3.6.4 Explanations and Semantics

7.3.6.4.1 Applications, DomainParticipants, and Contained Entities

Application Libraries are collections of Applications. Applications are in turn aggregations of DomainParticipants and their contained entities. Application Libraries are the top level elements of Building Block Applications.

7.3.6.4.1.1 Example (Non-normative)

```
{
    "name": "MyApplicationLibrary",
    "applications": [
        {
            "name": "MyApplication",
            "domain_participants": [
                {
                    "name": "MyParticipant",
                    "domain_ref": "BaseDomainLibraryName::BaseDomainName",
                    "publishers": [
                        {
                            "name": "MyPublisher",
                            "data_writers": [
                                {
                                    "name": "MySquareWriter",
                                    "topic_ref": "Square"
                                }
                            ]
                        }
                    ],
```

```
                    "subscribers": [
                        {
                            "name": "MySubscriber",
                            "data_readers": [
                                {
                                    "name": "MySquareReader",
                                    "topic_ref": "Square"
                                }
                            ]
                        }
                    ]
                }
            ]
        }
    ]
}
```

7.3.6.4.2 Using DomainParticipants defined in DomainParticipant Libraries

DomainParticipants defined in the context of an Application may inherit from a DomainParticipant defined in the context of a DomainParticipant Library using the **"base_name"** property, as specified in Clause 7.3.5.4.3.

7.3.6.4.2.1 Example (Non-normative)

```
{
    "name": "MyApplication",
    "domain_participants" : [
        {
            "name": "MyParticipant",
            "base_name": "BaseDomainParticipantLibraryName::BaseDomainParticipantName",
            ...
        }
    ]
}
```

7.3.7 Building Block Data Samples

7.3.7.1 Purpose

This block defines syntax to represent Data Samples that may be exchanged between different DDS applications in JSON format.

7.3.7.2 Dependencies with other Building Blocks

This building block has no dependencies on other building blocks.

7.3.7.3 Syntax

The following normative JSON schema file defines the syntax to represent DDS Data Samples and Sample Information:

- *dds-json_data_samples.schema.json*

Moreover, the following non-normative file contains an example on how to apply the aforementioned schema to represent Data Samples and Sample Information in JSON format:

- *dds-json_data_samples_example.json*

Because it is impossible to define a generic JSON schema file to represent Data Samples for all the possible Data Type combinations in DDS, *dds-json_data_samples.schema.json* defines just the syntax that is common to the representation of all Data Samples: the syntax to represent the Sample Information (i.e., the metadata portion of the sample), and the syntax to represent primitive types.

Therefore, the complete syntax to represent Data Samples is based on the mapping rules and JSON schema definitions specified in this building block, and the syntax to represent Sample Information specified in *dds-json_data_samples.schema.json*.

Implementers of this specification who may want to define and provide schema files to validate the syntax of Data Samples of user-defined data types shall generate JSON schema files following the rules specified in this building block, adding the syntax to define Sample Information defined in *dds-json_data_samples.schema.json*.

7.3.7.4 Explanations and Semantics

7.3.7.4.1 JSON Representation of Structures

Structures shall be represented as JSON objects including members of the structure as properties of the corresponding object. The name of the corresponding properties shall be the name of the structure members with no changes.

Unset optional members shall be omitted from the sample representation.

7.3.7.4.1.1 Example (Non-normative)

For a structured type defined in IDL as follows:

```
struct InnerStruct {
    long x;
    long y;
};
struct OuterStruct {
    long a;
    InnerStruct s;
};
```

The JSON representation of a sample would need to comply with the following schema:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "definitions": {
        "InnerStruct": {
            "type": "object",
            "properties": {
                "x": {
                    "type": "integer"
                },
                "y": {
                    "type": "integer"
                }
            }
        }
    },
    "type": "object",
    "properties": {
        "s": {
            "$ref": "#/definitions/InnerStruct"
        },
        "a": {
            "type": "integer"
        }
    }
}
```

For example:

```
{
    "a": 5,
    "s": {
        "x": 4,
        "y": 3
```

```
    }
}
```

7.3.7.4.2 JSON Representation of Unions

Unions shall be represented as JSON objects including the specific union case that was selected as a property. Therefore, the mapping is equivalent to that of a structure with the member selected by the union case (see Clause 7.3.7.4.1). The property name shall be the name of the original union member with no changes.

The JSON representation of a Union may optionally include the value of the discriminator field for reference. In that case, the discriminator shall be represented as a property of named **`"$discriminator"`**.

7.3.7.4.2.1 Example (Non-normative)

For a union type defined in IDL as follows:

```
union MyUnion switch(long) {
case 1:
    float x;
case 2:
    long y;
default:
    string z;
};
```

The JSON representation of a sample containing the union would need to comply with the following schema:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "oneOf": [
        {
            "properties": {
                "$discriminator": {
                    "type": "integer"
                },
                "x": {
                    "type": "number"
                }
            },
            "additionalProperties": false
        },
        {
            "properties": {
                "$discriminator": {
                    "type": "integer"
                },
                "y": {
                    "type": "integer"
                }
            },
            "additionalProperties": false
        },
        {
            "properties": {
                "$discriminator": {
                    "type": "integer"
                },
                "z": {
                    "type": "string"
                }
            },
            "additionalProperties": false
        }
    ]
```

```
}
```

For example:

```
{
    "$discriminator": 1,
    "x": 4.5
}
```

or

```
{
    "x": 4.5
}
```

7.3.7.4.3 JSON Representation of Sequences and Arrays

Sequences and arrays shall be represented as JSON arrays of the corresponding type. Sequence and array elements shall be represented as elements of the corresponding JSON array according to the mapping rules specified in this building block.

7.3.7.4.3.1 Example (Non-normative)

For a sequence defined in IDL as:

```
struct Coordinates {
    long x;
    long y;
};
struct OuterStruct {
    sequence<Coordinates> coordinates_sequence;
};
```

The JSON representation of a sample would need to comply with the following schema:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "definitions": {
        "Coordinates": {
            "type": "object",
            "properties": {
                "x": {
                    "type": "integer"
                },
                "y": {
                    "type": "integer"
                }
            }
        },
        "CoordinatesSeq": {
            "type": "array",
            "items": {
                "$ref": "#/definitions/Coordinates"
            }
        }
    },
    "type": "object",
    "properties": {
        "coordinates_sequence": {
            "$ref": "#/definitions/CoordinatesSeq"
        }
    }
}
```

For example:

```
[
    {
```

```
        "x": 1,
        "y": 15
    },
    {
        "x": 4,
        "y": 11
    }
]
```

7.3.7.4.4 JSON Representation of Maps

Maps shall be represented as JSON objects. Each map element shall become a property of the corresponding JSON object, using the string representation of the map element key as the property name, and the equivalent JSON representation of the map value as the property value.

In the case of signed and unsigned integer key types, the string representation shall present the integer value in base 10. For **`string`** and **`wstring`** key types, the value of the map key shall be the value of the string with no changes[1].

7.3.7.4.4.1 Example (Non-normative)

For example, samples of a structure containing maps represented in IDL as follows:

```
struct MyStruct {
    map<string,long> known_satellites;
    map<long,char> ascii_characters;
};
```

Would need to conform with the following schema:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties": {
        "known_satellites": {
            "type": "object",
            "properties": {
                "earth": {
                    "type": "integer"
                },
                "mars": {
                    "type": "integer"
                },
                ...
            }
        },
        "ascii_characters": {
            "type": "object",
            "properties": {
                "65": {
                    "type": "string",
                    "maxLength": 1
                },
                "97": {
                    "type": "string",
                    "maxLength": 1
                },
                ...
            }
        }
    }
}
```

[1] Clause 7.2.2.4.3 of [DDS-XTYPES] mandates compliant implementations to support map key types of signed and unsigned integer, string, and wide string type. The behavior for other key types is undefined and may not be portable; therefore, the string representation of key types other those expliclty listed in [DDS-XTYPES] is out of the scope of this specification.

For example:

```
{
    "known_satelites": {
        "earth": 1,
        "mars": 2,
    },
    "ascii_characters": {
        "65": "A",
        "97": "a"
    }
}
```

7.3.7.4.5 JSON Representation of Enums

Enums shall be represented as properties of integer or string type holding the value of the corresponding enumeration literal[2].

7.3.7.4.5.1 Example (Non-normative)

Samples of a structure containing an enum, represented in IDL as follows:

```
enum Weekday {
    @value(1) MONDAY,
    @value(2) TUESDAY,
    @value(3) WEDNESDAY,
    ...
};
struct MyStruct {
    Weekday wd;
};
```

Would need to conform with the following schema:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "definitions": {
        "Weekday": {
            "oneOf": [
                {
                    "type": "string",
                    "enum": [
                        "MONDAY",
                        "TUESDAY",
                        "WEDNESDAY",
                        ...
                    ]
                },
                {
                    "type": "integer"
                }
            ]
        }
    },
    "properties": {
        "wd": {
            "$ref": "#/definitions/Weekday"
        }
    }
}
```

[2] This enables implementers of this specification to select one of the two representations to encode the value of an **enum** depending on the use case. Implementations shall be capable of converting the string or integer value representing the corresponding enumeration literal into the corresponding internal representation accordingly.

For example:

```
{
    "wd": "MONDAY"
}
```

or

```
{
    "wd": 1
}
```

7.3.7.4.6 JSON Representation of Bitmasks

Bitmasks shall be represented as properties of integer type holding the value of the corresponding Bitmask.

7.3.7.4.7 JSON Representation of String Types

Strings and wide strings shall be represented as properties of string type holding the value of the corresponding string.

7.3.7.4.7.1 Example (Non-normative)

Samples of a structure containing strings, represented IDL as follows:

```
struct MyStruct {
    wstring a_string;
    string another_string;
};
```

Would need to conform with the following schema:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties": {
        "a_string": {
            "type": "string"
        },
        "another_string": {
            "type": "string"
        }
    }
}
```

For example:

```
{
    "a_string": "A string!",
    "another_string": "El r\u00EDo mi\u00F1o"
}
```

7.3.7.4.8 JSON Representation of Primitive Types

Primitive types shall be represented as properties of JSON objects or elements of JSON arrays, according to the mapping rules for the containing type specified in this building block. The type definition for each primitive type in the DDS type system is defined in Table 7.1.

Table 7.1: JSON Representation of Primitive Types

Type	JSON Schema	Example
`boolean`	`{ "type": "boolean" }`	`{ "my_boolean": true }`

Type	JSON Schema	Example

byte	{
	"type": "integer",
	"minimum": 0,
	"maximum": 255
	}
		{
		"my_byte": 1
		}

int8	{
	"type": "integer",
	"minimum": -127,
	"maximum": 128
	}
		{
		"my_int8": -3
		}

uint8	{
	"type": "integer",
	"minimum": 0,
	"maximum": 255
	}
		{
		"my_uint8": 2
		}

int16	{
	"type": "integer",
	"minimum": -32768,
	"maximum": 32767
	}
		{
		"my_int16": -32000
		}

uint16	{
	"type": "integer",
	"minimum": 0,
	"maximum": 65535
	}
		{
		"my_uint16": 64000
		}

int32	{
	"type": "integer",
	"minimum": -2147483648,
	"maximum": 2147483647
	}
		{
		"my_int32": -21000000
		}

uint32	{
	"type": "integer",
	"minimum": 0,
	"maximum": 4294967295
	}
		{
		"my_int32": 21000000
		}

int64	{
	"oneOf": [
	{
	"type": "integer",
	"minimum": -9007199254740991,
	"maximum": 9007199254740991
	},
	{
	"type": "string"
	}
	]
	}
		{
		"my_int64": -31321212111
		}
		
		{
		"my_int64":
		"-9007199254740992"
		}

uint64	{
	"type": "integer",
	"minimum": 0,
	"maximum": 9007199254740991
	}
		{
		"my_int64": 31321212111
		}
		
		{
		"my_int64":
		"9007199254740992"
		}

Type	JSON Schema	Example
float32	{ "type": "number" }	{ "my_float32": 3.14 }
float64	{ "type": "number" }	{ "my_float64": 3.14345 }
float128	{ "type": "string" }	{ "my_float128": "My4xNA==" }
char8	{ "type": "string" }	{ "my_char8": "a" }
char16	{ "type": "string" }	{ "my_char16": "\u007E" } { "my_char16": "a" }

As shown in Table 7.1, values of most DDS primitive types can be represented using simply native JSON types. However, the following primitive types require special mapping rules:

- **byte** values shall be represented as properties of integer type in the range [0, 255] using base 10.
- **int64** values in the range [$-2^{53} + 1$, $2^{53} - 1$] shall be represented as properties of integer type. Valid **int64** values outside that range shall be represented as strings including the numeric value in base 10[3].
- **uint64** values in the range [0, $2^{53} - 1$] shall be represented as properties of integer type. Valid **uint64** values outside that range shall be represented as strings including the numeric value in base 10[3].
- **float128** values shall be represented as properties of string type encoding the value of the **float128** member using base64 according to [RFC-4648].

Numeric values, such as **Infinity**, **-Infinity**, and **NaN**, which as stated in [ECMA-404] cannot be represented as sequences of digits, shall be represented using the following JSON strings: **"inf"**, **"-inf"**, and **"nan"**.

[3] This mapping is consistent with the recommendations of [RFC-8259] and the I-JSON profile defined in [RFC-7493]. The latter states that "an I-JSON sender cannot expect an integer whose absolute value is greater than 9007199254740991 (i.e., that is outside the range [$-2^{53} + 1$, $2^{53} - 1$]) as an exact value."

This page intentionally left blank.