OPC UA/DDS Gateway

**FTF Beta 1 Version 1.0**

OMG Document Number: ptc/2019-02-2588

Normative Reference: [http://www.omg.org/spec/DDS-OPCUA/1.0](http://www.omg.org/spec/DDS-OPCUA/1.0)

Associated Normative Machine Consumable Files:


Associated Non-Normative Machine Consumable Files:


This OMG document replaces the submission document (mars/18-03-01, Alpha). It is an OMG Adopted Beta-Specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to issues@.omg.org by September 30, 2018.

You may view the pending issues for this specification from the OMG revision issues web page: [https://issues.omg.org/issues/lists](https://issues.omg.org/issues/lists).

The FTF Recommendation and Report for this specification will be published in March 2019. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.
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
MERCHANTIBILITY 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, 140 Kendrick Street, Needham, MA 02494, 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: http://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.

OPC UA/DDS Gateway 1.0
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 http://www.omg.org, under Documents, Report a Bug/Issue (http://issues.omg.org/issues/create-new-issue).
# Table of Contents

1 Scope ................................................................................................................................. 1

2 Conformance ....................................................................................................................... 1

3 Normative References ......................................................................................................... 2

4 Terms and Definitions ......................................................................................................... 4

5 Symbols ............................................................................................................................... 4

6 Additional Information ....................................................................................................... 5
   6.1 Acknowledgements ......................................................................................................... 5

7 OPC UA/DDS Gateway Overview (non-normative) ............................................................... 7
   7.1 OPC Unified Architecture (OPC UA) .............................................................................. 7
      7.1.1 OPC UA AddressSpace ......................................................................................... 7
      7.1.2 OPC UA Services ................................................................................................. 8
   7.2 Data Distribution Service (DDS) ................................................................................... 9
      7.2.1 DDS Global Data Space ....................................................................................... 9
      7.2.2 Remote Procedure Call over DDS (DDS-RPC) ..................................................... 10
   7.3 Bridging OPC UA and DDS ........................................................................................ 10

8 OPC UA to DDS Bridge ...................................................................................................... 13
   8.1 Overview (non-normative) ............................................................................................ 13
   8.2 OPC UA Type System Mapping ................................................................................... 13
      8.2.1 Built-in Primitive Types ..................................................................................... 14
      8.2.2 Built-in Complex Types ..................................................................................... 15
   8.3 OPC UA Service Sets Mapping .................................................................................... 18
      8.3.1 Standard DataTypes and NodeClasses Mapping .................................................. 18
      8.3.2 View Service Set ............................................................................................... 25
      8.3.3 Query Service Set ............................................................................................ 26
      8.3.4 Attribute Service Set ....................................................................................... 27
      8.3.5 Method Service Set ........................................................................................... 32
      8.3.6 Implementation Considerations .......................................................................... 32
   8.4 OPC UA Subscription Model Mapping ......................................................................... 34
      8.4.1 Overview (non-normative) .................................................................................. 34
      8.4.2 OPC UA Subscription Mapping .......................................................................... 37
      8.4.3 OPC UA Subscription Mapping Behavior ............................................................ 46
      8.4.4 Implementation Considerations .......................................................................... 51

9 DDS to OPC UA Bridge ...................................................................................................... 53
   9.1 Overview (non-normative) ............................................................................................ 53
   9.2 DDS Type System Mapping ........................................................................................ 53
      9.2.1 Primitive Types ................................................................................................... 54
      9.2.2 String Types ....................................................................................................... 58
      9.2.3 Enumerated Types .............................................................................................. 61
      9.2.4 Aggregated Types .............................................................................................. 70
      9.2.5 Collection Types ............................................................................................... 80
      9.2.6 Nested Types ..................................................................................................... 102
      9.2.7 Alias Types ....................................................................................................... 102
      9.2.8 Keyed Types ..................................................................................................... 102
   9.3 DDS Global Data Space Mapping ................................................................................. 103
      9.3.1 Overview (non-normative) ................................................................................. 103
      9.3.2 Representing DDS Domains in OPC UA ............................................................. 104

OPC UA/DDS Gateway 1.0  v
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 7.1:</td>
<td>OPC UA Metamodel.</td>
<td>8</td>
</tr>
<tr>
<td>Figure 7.2:</td>
<td>DCPS Conceptual Model.</td>
<td>10</td>
</tr>
<tr>
<td>Figure 7.3:</td>
<td>OPC UA/DDS Gateway Concept.</td>
<td>11</td>
</tr>
<tr>
<td>Figure 8.1:</td>
<td>OPC UA to DDS Bridge Overview.</td>
<td>13</td>
</tr>
<tr>
<td>Figure 8.2:</td>
<td>OPC UA Subscription Mapping Overview.</td>
<td>38</td>
</tr>
<tr>
<td>Figure 8.3:</td>
<td>OPC UA Input Definition.</td>
<td>39</td>
</tr>
<tr>
<td>Figure 8.4:</td>
<td>DDS Output Definition.</td>
<td>42</td>
</tr>
<tr>
<td>Figure 8.5:</td>
<td>Input/Output Mapping Definition.</td>
<td>44</td>
</tr>
<tr>
<td>Figure 9.1:</td>
<td>DDS to OPC UA Bridge Overview.</td>
<td>53</td>
</tr>
<tr>
<td>Figure 9.2:</td>
<td>Primitive Types Mapping to OPC UA—Integer Types.</td>
<td>54</td>
</tr>
<tr>
<td>Figure 9.3:</td>
<td>Primitive Types Mapping to OPC UA—Floating Point Types.</td>
<td>55</td>
</tr>
<tr>
<td>Figure 9.4:</td>
<td>Primitive Types Mapping to OPC UA—Boolean, Byte, and Char Types.</td>
<td>55</td>
</tr>
<tr>
<td>Figure 9.5:</td>
<td>Example of Primitive Type Mapping to OPC UA.</td>
<td>57</td>
</tr>
<tr>
<td>Figure 9.6:</td>
<td>String Types Mapping to OPC UA.</td>
<td>59</td>
</tr>
<tr>
<td>Figure 9.7:</td>
<td>Example of String Type Mapping to OPC UA.</td>
<td>60</td>
</tr>
<tr>
<td>Figure 9.8:</td>
<td>Enumeration Types Mapping to OPC UA.</td>
<td>62</td>
</tr>
<tr>
<td>Figure 9.9:</td>
<td>Example of Enumeration Type Mapping to OPC UA.</td>
<td>63</td>
</tr>
<tr>
<td>Figure 9.10:</td>
<td>Bitmap Types Mapping to OPC UA.</td>
<td>66</td>
</tr>
<tr>
<td>Figure 9.11:</td>
<td>Example of Bitmap Type Mapping to OPC UA.</td>
<td>68</td>
</tr>
<tr>
<td>Figure 9.12:</td>
<td>Structure Types Mapping to OPC UA.</td>
<td>72</td>
</tr>
<tr>
<td>Figure 9.13:</td>
<td>Example of Structure Type Mapping to OPC UA.</td>
<td>74</td>
</tr>
<tr>
<td>Figure 9.14:</td>
<td>Union Types Mapping to OPC UA.</td>
<td>77</td>
</tr>
<tr>
<td>Figure 9.15:</td>
<td>Example of Union Type Mapping to OPC UA.</td>
<td>79</td>
</tr>
<tr>
<td>Figure 9.16:</td>
<td>Array of Primitive or String Types Mapping to OPC UA.</td>
<td>82</td>
</tr>
<tr>
<td>Figure 9.17:</td>
<td>Array of Enumerations Mapping to OPC UA.</td>
<td>83</td>
</tr>
<tr>
<td>Figure 9.18:</td>
<td>Array of Bitmasks Mapping to OPC UA.</td>
<td>84</td>
</tr>
<tr>
<td>Figure 9.19:</td>
<td>Array of Structures Mapping to OPC UA.</td>
<td>85</td>
</tr>
<tr>
<td>Figure 9.20:</td>
<td>Array of Unions Mapping to OPC UA.</td>
<td>88</td>
</tr>
<tr>
<td>Figure 9.21:</td>
<td>Array of Collection Types Mapping to OPC UA.</td>
<td>89</td>
</tr>
<tr>
<td>Figure 9.22:</td>
<td>Sequence of Primitive or String Types Mapping to OPC UA.</td>
<td>91</td>
</tr>
<tr>
<td>Figure 9.23:</td>
<td>Sequence of Enumerations Mapping to OPC UA.</td>
<td>92</td>
</tr>
<tr>
<td>Figure 9.24:</td>
<td>Sequence of Bitmasks Variable Definition.</td>
<td>93</td>
</tr>
<tr>
<td>Figure 9.25:</td>
<td>Sequence of Structures Mapping to OPC UA.</td>
<td>94</td>
</tr>
<tr>
<td>Figure 9.26:</td>
<td>Sequence of Unions Mapping to OPC UA.</td>
<td>96</td>
</tr>
<tr>
<td>Figure 9.27:</td>
<td>Sequence of Collection Types Mapping to OPC UA.</td>
<td>97</td>
</tr>
<tr>
<td>Figure 9.28:</td>
<td>Map Types Mapping to OPC UA.</td>
<td>99</td>
</tr>
<tr>
<td>Figure 9.29:</td>
<td>Example of Map Type Mapping to OPC UA.</td>
<td>100</td>
</tr>
<tr>
<td>Figure 9.30:</td>
<td>DDS Domain Mapping to OPC UA.</td>
<td>104</td>
</tr>
<tr>
<td>Figure 9.31:</td>
<td>DDS Topic Mapping to OPC UA.</td>
<td>108</td>
</tr>
<tr>
<td>Figure 9.32:</td>
<td>DDS Instance Mapping to OPC UA.</td>
<td>113</td>
</tr>
</tbody>
</table>
Table of Tables

Table 2.1: Conformance Points.................................................................1
Table 5.1: Acronyms...............................................................................4
Table 8.1: Mapping of OPC UA Primitive Types to DDS..........................14
Table 8.2: Mapping of OPC UA Non-Primitive Built-in Types to DDS........15
Table 8.3: Mapping of OPC UA Standard DataTypes and NodeClasses to DDS. 18
Table 8.4: Mapping of Types Specific to the View Service Set..................25
Table 8.5: Mapping of Types Specific to the Query Service Set.................26
Table 8.6: Mapping of Types Specific to the Attribute Service Set.............28
Table 8.7: Mapping of Types Specific to the Method Service Set...............32
Table 8.8: Subscription Mapping Configuration......................................38
Table 8.9: OPC UA Input Definition......................................................39
Table 8.10: OPC UA Connection Definition.........................................40
Table 8.11: OPC UA Subscription Protocol Definition..............................40
Table 8.12: OPC UA MonitoredItem Definition......................................41
Table 8.13: DDS Output Definition......................................................42
Table 8.14: DDS DomainParticipant Definition......................................43
Table 8.15: Input/Output Mapping Definition.........................................45
Table 8.16: Simplified Mapping of OPC UA Variant Type to DDS Types.......48
Table 9.1: Primitive Type Variable Definition.........................................55
Table 9.2: OPC UA Built-in Types Equivalent to DDS Primitive Types.........56
Table 9.3: Example of Int32 Variable Definition.......................................57
Table 9.4: String8 (String) Variable Definition........................................59
Table 9.5: String16 (Wide String) Variable Definition..............................59
Table 9.6: Example of String Variable Definition......................................60
Table 9.7: Enumeration DataType Definition.........................................61
Table 9.8: Enumeration Variable Definition...........................................62
Table 9.9: Example of Enumeration DataType Definition........................63
Table 9.10: Example of Enumeration Variable Definition.........................64
Table 9.11: Bitmask DataType Definition................................................65
Table 9.12: Bitmask Variable Definition..................................................66
Table 9.13: Example of Bitmask DataType Definition.................................67
Table 9.14: Example of Bitmask Variable Definition..................................68
Table 9.15: Structure DataType Definition...............................................70
Table 9.16: Structure VariableType Definition.........................................71
Table 9.17: Example of Structure DataType Definition..............................72
Table 9.18: Example of Structure VariableType Definition.........................73
Table 9.19: Example of Structure Variable Definition................................73
Table 9.20: Union DataType Definition..................................................75
Table 9.21: Union Variable Definition....................................................76
Table 9.22: Example of Union DataType Definition..................................77
Table 9.23: Example of Union Variable Definition....................................77
Table 9.24: Array of Primitive or String Type Variable Definition..............79
Table 9.25: Array of Enumerations Variable Definition..............................80
Table 9.26: Array of Bitmasks Variable Definition.....................................81
Table 9.27: Array of Structures Variable Definition...................................84
Table 9.28: Array of Unions Variable Definition.......................................85
Table 9.29: Array of Collection Types Object Definition............................86
Table 9.30: Collection Variable or Object Definition – Arrays of Collections. 86
Table 9.31: Example Array Variable Definition.........................................87
Table 9.32: Sequence of Primitive or String Types Variable Definition........88
Table 9.33: Sequence of Enumerations Variable Definition.........................89
Table 9.34: Sequence of Bitmasks Variable Definition...............................90
Table 9.35: Sequence of Structures Variable Definition.............................91
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 http://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:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

• CORBA/IIOP
• Data Distribution Services
• Specialized CORBA

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

• UML, MOF, CWM, XMI
• UML Profiles

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

• CORBAServices
• CORBAFacilities

OMG Domain Specifications
CORBA Embedded Intelligence Specifications

CORBA Security Specifications

Signal and Image Processing Specifications

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
109 Highland Avenue
Needham, MA 02494
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

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman/Liberation Serif – 10 pt.: Standard body text

Helvetica/Arial – 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.


Helvetica/Arial – 10 pt: Exceptions

NOTE: Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification via the report form at:

http://issues.omg.org/issues/create-new-issue
This page intentionally left blank.
1 Scope

Data Distribution Service (DDS) is a family of standards from the Object Management Group (OMG) that provide connectivity, interoperability, and portability for Industrial Internet, cyber-physical, and mission-critical applications.

The DDS connectivity standards cover Publish-Subscribe (DDS), Service Invocation (DDS-RPC), Interoperability (DDS-RTPS), Information Modeling (DDS-XTYPES), Security (DDS-SECURITY), as well as programing APIs for C, C++, Java and other languages.

The OPC Unified Architecture (OPC UA) is an information exchange standard for Industrial Automation and related systems created by the OPC Foundation. The OPC UA standard provides an Addressing and Information Model for Data Access, Alarms, and Service invocation layered over multiple transport-level protocols such as Binary TCP and Web-Services.

DDS and OPC UA exhibit significant deployment similarities:

- Both enable independently developed applications to interoperate even when those applications come from different vendors, use different programming languages, or run on different platforms and operating systems.
- Both have significant traction within Industrial Automation systems.
- Both define standard protocols built on top of the TCP/ UDP/IP Internet stacks.

The two technologies may coexist within the same application domains; however, while there are solutions that bridge between DDS and OPC UA, these are based on custom mappings and cannot be relied to work across vendors and products.

This specification overcomes this situation by defining a standard, vendor-independent, configurable gateway that enables interoperability and information exchange between systems that use DDS and systems that use OPC UA.

2 Conformance

This specification defines a set of building blocks that are grouped into four conformance points:

- OPC UA to DDS Mapping Basic Conformance
- OPC UA to DDS Mapping Complete Conformance
- DDS to OPC UA Mapping Basic Conformance
- OPC UA to DDS Mapping Complete Conformance

Table 2.1 defines each conformance point and lists the building blocks they are built upon.

<table>
<thead>
<tr>
<th>Conformance Point</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPC UA to DDS Mapping Basic Conformance</td>
<td>Constructs an OPC UA/DDS Gateway that allows DDS applications to subscribe to data in the AddressSpace of different OPC UA Servers. Conformance with this point requires the implementation of the following building blocks:</td>
</tr>
<tr>
<td></td>
<td>• OPC UA Type System Mapping</td>
</tr>
<tr>
<td>Conformance Point</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------</td>
</tr>
<tr>
<td>• OPC UA Subscription Model Mapping</td>
<td></td>
</tr>
<tr>
<td>OPC UA to DDS Mapping Complete Conformance</td>
<td>Constructs an OPC UA/DDS Gateway that allows DDS applications to subscribe, browse, and manage data in the AddressSpace of different OPC UA Servers. Conformance with this point requires the implementation of:</td>
</tr>
<tr>
<td>• OPC UA to DDS Mapping Basic Conformance</td>
<td></td>
</tr>
<tr>
<td>• OPC UA Service Sets Mapping</td>
<td></td>
</tr>
<tr>
<td>DDS to OPC UA Mapping Basic Conformance</td>
<td>Constructs an OPC UA/DDS Gateway that allows OPC UA clients to browse, read, write, and subscribe to information in the DDS Global Data Space. Conformance with this point requires the implementation of the following building blocks:</td>
</tr>
<tr>
<td>• DDS Type System Mapping</td>
<td></td>
</tr>
<tr>
<td>• DDS Global Data Space Mapping (except sub clause 9.3.4.4 Reading Historical Data from Instance Nodes)</td>
<td></td>
</tr>
<tr>
<td>DDS to OPC UA Mapping Complete Conformance</td>
<td>Constructs an OPC UA/DDS Gateway that allows OPC UA clients to browse, read, write, and subscribe to information in the DDS Global Data Space, Services. Additionally, it allows OPC UA clients to access Historical Data. Conformance with this point requires the implementation of:</td>
</tr>
<tr>
<td>• DDS to OPC UA Mapping Basic Conformance</td>
<td></td>
</tr>
<tr>
<td>• Reading Historical Data from Instance Nodes</td>
<td></td>
</tr>
</tbody>
</table>

OPC UA to DDS and DDS to OPC UA conformance points may be combined in implementations of the OPC UA/DDS Gateway that provide bi-directional communication between OPC UA and DDS applications. For example:

- Implementations conforming to OPC UA to DDS Mapping Basic Conformance and DDS to OPC UA Mapping Basic Conformance provide basic bi-directional communication between OPC UA and DDS applications.
- Implementations conforming to OPC UA to DDS Mapping Complete Conformance and DDS to OPC UA Mapping Complete Conformance provide complete bi-directional communication between OPC UA and DDS applications.

## 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.


DDSOPCU-5: Update dds-opcua_model.xmi and normative references to most recent versions


DDSOPCU-1: Update Schema Files and XML files to Use DDS-XML 1.0


DDSOPCU-5: Update dds-opcua_model.xmi and normative references to most recent versions


4 Terms and Definitions

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

**DDS**

Data Distribution Service (DDS) is a family of standards from the Object Management Group (OMG, [http://www.omg.org](http://www.omg.org)) that provide connectivity, interoperability and portability for Industrial Internet, cyber-physical, and mission-critical applications. The DDS connectivity standards cover Publish-Subscribe (DDS), Service Invocation (DDS-RPC), Interoperability (DDSI-RTPS), Information Modeling (DDS-XTYPES), Security (DDS-Security), as well as programming APIs for C, C++, Java and other languages.

**DDS Domain**

Represents a global data space. It is a logical scope (or “address space”) for Topic and Type definitions. Each Domain is uniquely identified by an integer Domain ID. Domains are completely independent from each other. For two DDS applications to communicate with each other they must join the same DDS Domain.

**DDS DomainParticipant**

A DomainParticipant is the DDS Entity used by an application to join a DDS Domain. It is the first DDS Entity created by an application and serves as a factory for other DDS Entities. A DomainParticipant can join a single DDS Domain. If an application wants to join multiple DDS Domains, then it must create corresponding DDS DomainParticipant entities, one per domain.

**Mapping**

Specifies how to implement a DDS or an OPC UA feature with a specific technology [OPCUA-06].

**OPC UA**

OPC Unified Architecture (OPC UA) is an information exchange standard for Industrial Automation and related systems created by the OPC Foundation ([http://www.opcfoundation.org](http://www.opcfoundation.org)). The OPC UA standard provides an Addressing and Information Model for Data Access, Alarms, and Service invocation, layered over multiple transport-level protocols such as Binary TCP and Web-Services.

5 Symbols

The following acronyms are used in this specification.

<table>
<thead>
<tr>
<th>Acronyms</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPS</td>
<td>Data-Centric Publish-Subscribe</td>
</tr>
<tr>
<td>DDS</td>
<td>Data Distribution Service</td>
</tr>
<tr>
<td>GDS</td>
<td>Global Data Space</td>
</tr>
<tr>
<td>Acronyms</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>RTPS</td>
<td>Real-Time Publish-Subscribe Protocol</td>
</tr>
<tr>
<td>UA</td>
<td>Unified Architecture</td>
</tr>
<tr>
<td>XTYPES</td>
<td>eXtensible and dynamic topic TYPES (for DDS)</td>
</tr>
</tbody>
</table>

### 6 Additional Information

#### 6.1 Acknowledgements

The following companies submitted this specification:

- Real-Time Innovations, Inc.
- PrismTech Ltd
- Twin Oaks Computing, Inc.
- eProsima, Inc.
7  OPC UA/DDS Gateway Overview (non-normative)

7.1  OPC Unified Architecture (OPC UA)

OPC UA defines a pure client-server architecture, where Clients access the AddressSpace of a Server by means of a set of standard Services. This clause provides an overview of the OPC UA AddressSpace and Service Sets focusing on the aspects that are important for building a bridge between OPC UA and DDS.

[OPCUA-01] provides a more general purpose overview of OPC UA and the different parts of the specification.

7.1.1  OPC UA AddressSpace

The OPC UA AddressSpace model provides a mechanism to describe the entities that exist in a distributed system. It is defined in [OPCUA-03] using UML as a meta-model that may be exposed by any OPC UA Server.

The AddressSpace is composed of a set of Nodes connected by References. Figure 7.1 depicts the different NodeClasses defined in the OPC UA standard and their relationship with References.

- **BaseNodeClass**—The abstract class BaseNodeClass contains the set of Attributes that are common to all NodeClasses including a NodeClass enumeration attribute that indicates which concrete class is actually instantiated, and a NodeId that uniquely identifies a Node anywhere in the system. Note that relationships between Nodes are defined by means of the NodeId value (similarly to a foreign key in a relational data model).

- **ReferenceType**—ReferenceTypes define the nature of references (relationship between Nodes). Clause 7 of [OPCUA-03] defines a set of standard ReferenceTypes, which are widely used in OPC UA applications. Other parts of the OPC UA family of standards define additional ReferenceTypes by instantiating the ReferenceType NodeClass. It is important to note that References are not NodeClasses and they do not appear as such in the AddressSpace of OPC UA Servers.

- **View**—Nodes of the View class allow the selection of a subset of the AddressSpace. The entire AddressSpace is the default view. Each node in a view may contain only a subset of its References, as defined by the creator of the view.

- **Object**—Nodes of the Object NodeClass represent real-life objects in a system. Examples of Objects are devices, controllers dealing with multiple devices, segments containing multiple controllers, and plants consisting of multiple segments.

- **ObjectType**—Nodes of the ObjectType NodeClass provide type definitions for Objects. In other words, Objects are defined by ObjectTypes, and each node of Object class includes a HasTypeDefinition Reference to an ObjectType.

- **Variable**—Nodes of the Variable NodeClass represent simple or complex values. Depending on their constraints, Variables are defined as either Properties or DataTypes of other Nodes. Variables may be simple or complex. Simple Variable objects refer to predefined DataTypes as found in [OPCUA-06].

- **VariableType**—Nodes of the VariableTypes NodeClass provide type definitions for Variables. In other words, Variables are defined by VariableTypes, and each node of the Variable includes a HasTypeDefinition Reference to a VariableType.

- **Method**—Nodes of the Method NodeClass define functions that are invoked using the Call Service defined in [OPCUA-04].

- **DataType**—Nodes of the DataType NodeClass describe the syntax of a Variable’s value. DataTypes can be simple or complex.
7.1.2 OPC UA Services

In a nutshell, OPC UA Services are Remote Procedure Calls (RPC) that Client applications can invoke to browse the AddressSpace of a Server, read/write data, and configure subscriptions. OPC UA’s complete Service Set is defined in [OPCUA-04].

For the purpose of building a bridge between OPC UA and DDS the following Service Sets apply:

- **View Service Set**—Provides Clients with Services to navigate the AddressSpace or a View—a subset—of the AddressSpace of an OPC UA Server. These include the Browse, and BrowseNext services.

- **Query Service Set**—Provides Clients with Services to access information about the OPC UA Server. These include the QueryFirst and QueryNext services.

Figure 7.1: OPC UA Metamodel
• **Attribute Service Set**—Provides Clients with Services to Attributes that are part of a Nodes. For example, it allows Clients to read the value of a Variable Node using the Read Service, update the value of a Variable Node using the Write Service, or perform operations on historical values or events using the HistoryRead or HistoryUpdate services.

• **Method Service Set**—Provides Clients with the Call Service, which is used to invoke OPC UA Methods.

• **Subscription Service Set**—Provides Clients with a mechanism to receive notifications from the Server on a group of MonitoredItems. Unlike in DDS, where subscriptions are configured on a per-Topic bases (which decouples information producers from information consumers in time and space, and allows efficient one-to-many and many-to-many communications), OPC UA Subscriptions are server-to-client (i.e., one-to-one). As a result, a Client is tightly coupled to a Server. In other words, Clients configure their own Subscriptions on the Server and cannot share them with other Clients.

• **MonitoredItems Service Set**—Provides Clients with Services to configure the data and Events they wish to subscribe to. MonitoredItems are created in the context of a Subscription, which is used to push Notifications to the Client.

OPC UA provides also Service Sets to manage and control connections between OPC UA Clients and Servers. While these services need not be exposed to DDS applications—because they have no role in the OPC UA to DDS end-to-end interactions—they shall be implemented by the OPC UA Clients and Servers embedded into the OPC UA/DDS Gateway (see sub clause 7.3).

• **Discovery Service Set**—Provides Clients with Services to discover Endpoints they can use to establish a SecureChannel.

• **SecureChannel Service Set**—Provides Clients with Services to open a communication channel to exchange Messages with the Server.

• **Session Service Set**—Provides Clients with Services to create an application-layer connection once a SecureChannel has been created.

• **NodeManagement Service Set**—Provides Clients with Services to modify the AddressSpace of a Server. This Service Set needs not be implemented by the OPC UA/DDS Gateway.

### 7.2 Data Distribution Service (DDS)

DDS is based on a data-centric publish-subscribe (DCPS) communication model, where information producers and information consumers are decoupled in time and space and exchange information by means of a set of Topics. This enables seamless one-to-many and many-to-many communication.

#### 7.2.1 DDS Global Data Space

The DDS DCPS model is built upon the concept of a Global Data Space (GDS) that is accessible to all interested applications. DDS applications that are interested in contributing information to the GDS become Publishers and DDS applications interested in portions of the GDS become Subscribers. Each time a Publisher posts new data into the Global Data Space, the DDS middleware propagates the information to the corresponding Subscribers [DDS].

The information that Publishers and Subscribers exchange in the Global Data Space is referred to as Topics, which uniquely identify the data items in the Global Data Space. Each Topic is associated with a Type, which provides information on how to manipulate the data, providing a level of type safety.

Lastly, the Global Data Space is divided into different logical divisions called Domains. DDS applications may participate in different Domains using different DomainParticipants. Likewise, DomainParticipants may create different DataWriters and DataReaders to publish and subscribe to different Topics on a certain Domain. Figure 7.2
provides an overview of the DCPS Model and shows the different DDS Entities that enable applications to participate in the Global Data Space.

### 7.2.2 Remote Procedure Call over DDS (DDS-RPC)

While the publish-subscribe communications model makes DDS extremely powerful and scalable for one-to-many and many-to-many communications, it makes it cumbersome to implement request-reply interactions and RPC invocations such as OPC UA’s.

To overcome this limitation, the DDS family of standards includes the RPC over DDS Specification [DDS-RPC], which defines a standard RPC framework using the basic building blocks of DDS (e.g., Topics, Types, DataWriters, and DataReaders) to provide request-reply semantics.

The [IDL] specification provides syntax to represent services and interfaces and the [DDS-RPC] specification provides the corresponding mapping of that syntax to actual building blocks to implement the DDS services and interfaces.

### 7.3 Bridging OPC UA and DDS

The goal of this specification is to define a standard, vendor-independent, configurable gateway to enable seamless interoperability and information exchange between systems that use DDS and systems that use OPC UA.

An important use-case that would greatly benefit from a standards-based gateway is the use of DDS to integrate OPC UA applications and subsystems (see Figure 7.3). In this scenario, individual applications and components, which expose their data and services via OPC UA, are integrated into larger systems for monitoring and control using DDS.
These systems would benefit from OPC UA’s familiar Industrial Automation information models while benefiting from DDS’ scalability, performance, QoS, and Global Data Space abstractions.

**DDSOPCU-3: Improve readability of a number of figures**

![Diagram of OPC UA/DDS Gateway Concept](image)

**Figure 7.3: OPC UA/DDS Gateway Concept**

An OPC UA/DDS Gateway capable of providing such functionality must implement two different bridges:

- **OPC UA to DDS Bridge**, which enables DDS applications to interact with the AddressSpace of different OPC UA Servers using native constructs,
- **DDS to OPC UA Bridge**, which enables OPC UA Clients to participate as first-class citizens in the DDS Global Data Space.

Additionally, the OPC UA/DDS Gateway must provide a set of configuration files to allow users to tune the behavior and mappings of the Gateway to their needs.

It is important to note that this specification does not mandate any specific architecture for the OPC UA/DDS Gateway, although it describes an implementation based on the use of built-in OPC UA Clients and Servers and DDS Entities. Instead, it provides a set of building blocks that enable implementers of this specification to construct an interoperable product.
8 OPC UA to DDS Bridge

This chapter defines the OPC UA to DDS Bridge, which enables DDS applications to browse, read, and manage information in the AddressSpace of different OPC UA Servers. In other words, it enables DDS applications to communicate with OPC UA Servers using DDS native constructs.

8.1 Overview (non-normative)

Figure 8.1 shows an example OPC UA/DDS Gateway implementing the OPC UA to DDS Bridge.

8.2 OPC UA Type System Mapping

OPC UA leverages a collection of built-in types to construct structures, arrays, and messages.

Sub clause 5.1.2 of [OPCUA-06] defines the complete set of built-in types and assigns an ID to each of them. The set of OPC UA built-in types can be represented as the following enumeration in IDL syntax:

```
module OMG { module DDSOPCUA { module OPCUA2DDS {
enum BuiltInTypeKind {
```
The OPC UA built-in types listed above include both primitive types and complex types. The mapping of primitive types to DDS is described in sub clause 8.2.1 and the mapping of complex types is described in sub clause 8.2.2. These mappings are also available in a separate normative machine-readable IDL file named dds-opcua_builtin_types.idl, which is provided with this specification.

### 8.2.1 Built-in Primitive Types

Table 8.1 shows the correspondence between the different built-in OPC UA primitive types and DDS types. The mapping provides both the generic [DDS-XTYPES] equivalent type and its corresponding [IDL] representation.

<table>
<thead>
<tr>
<th>OPC UA Built-in Type</th>
<th>DDS Type</th>
<th>IDL Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>SByte</td>
<td>Byte</td>
<td>int8</td>
</tr>
<tr>
<td>Byte</td>
<td>Byte</td>
<td>uint8</td>
</tr>
<tr>
<td>Int16</td>
<td>Int16</td>
<td>int16</td>
</tr>
<tr>
<td>UInt16</td>
<td>UInt16</td>
<td>uint16</td>
</tr>
<tr>
<td>Int32</td>
<td>Int32</td>
<td>int32</td>
</tr>
<tr>
<td>UInt32</td>
<td>UInt32</td>
<td>uint32</td>
</tr>
<tr>
<td>Int64</td>
<td>Int64</td>
<td>int64</td>
</tr>
<tr>
<td>UInt64</td>
<td>UInt64</td>
<td>uint64</td>
</tr>
</tbody>
</table>
There is almost a one-to-one correspondence between these types. The only exception are OPC UA SByte and Byte types, which represent signed and unsigned 8-bit integers respectively. [DDS-XTYPES] does not define an 8-bit signed integer; therefore, they are both mapped to DDS Bytes\(^1\). Nevertheless, these types can always be expressed in [IDL], which provides the equivalent int8 and uint8 types.

### 8.2.2 Built-in Complex Types

Table 8.2 maps the OPC UA non-primitive built-in types to IDL.

**Table 8.2: Mapping of OPC UA Non-Primitive Built-in Types to DDS**

<table>
<thead>
<tr>
<th>OPC UA Built-in Type</th>
<th>DDS Type (IDL Equivalent) (^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime</td>
<td>int64</td>
</tr>
<tr>
<td>Guid</td>
<td>struct Guid {</td>
</tr>
<tr>
<td></td>
<td>uint32 data1;</td>
</tr>
<tr>
<td></td>
<td>uint16 data2;</td>
</tr>
<tr>
<td></td>
<td>uint16 data3;</td>
</tr>
<tr>
<td></td>
<td>octet data4[8];</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>ByteString</td>
<td>sequence&lt;octet&gt;</td>
</tr>
<tr>
<td>XmlElement</td>
<td>string</td>
</tr>
<tr>
<td>NodeId</td>
<td>enum NodeIdentifierKind {</td>
</tr>
<tr>
<td></td>
<td>NODEID_NUMERIC,</td>
</tr>
<tr>
<td></td>
<td>NODEID_STRING,</td>
</tr>
<tr>
<td></td>
<td>NODEID_GUID,</td>
</tr>
<tr>
<td></td>
<td>NODEID_OPAQUE</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>@nested</td>
</tr>
<tr>
<td></td>
<td>union NodeIdentifierType switch (NodeIdentifierKind) {</td>
</tr>
<tr>
<td></td>
<td>case NUMERIC_NODE_ID:</td>
</tr>
<tr>
<td></td>
<td>uint32 numeric_id;</td>
</tr>
<tr>
<td></td>
<td>case STRING_NODE_ID:</td>
</tr>
<tr>
<td></td>
<td>string String_id;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>case GUID_NODE_ID:</td>
</tr>
<tr>
<td></td>
<td>Guid guid_id;</td>
</tr>
<tr>
<td></td>
<td>case OPAQUE_NODE_ID:</td>
</tr>
<tr>
<td></td>
<td>ByteString opaque_id;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>struct NodeId {</td>
</tr>
<tr>
<td></td>
<td>uint16 namespace_index;</td>
</tr>
<tr>
<td></td>
<td>NodeIdentifierType identifier_type;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

---

\(^1\) The addition of types Int8 and Uint8 is planned for the next revision of the [DDS-XTYPES] specification.

\(^2\) All these types appear inside the IDL module OMG::DDSOPCUA::OPCUA2DDS.
<table>
<thead>
<tr>
<th>OPC UA Built-in Type</th>
<th>DDS Type (IDL Equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExpandedNodeId</td>
<td>struct ExpandedNodeId : NodeId {</td>
</tr>
<tr>
<td></td>
<td>string namespace_uri;</td>
</tr>
<tr>
<td></td>
<td>uint32 server_index;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td>StatusCode</td>
<td>uint32</td>
</tr>
<tr>
<td>QualifiedName</td>
<td>struct QualifiedName {</td>
</tr>
<tr>
<td></td>
<td>uint16 namespace_index;</td>
</tr>
<tr>
<td></td>
<td>string name; // Restricted to 512 characters</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td>LocalizedText</td>
<td>@mutable struct LocalizedText {</td>
</tr>
<tr>
<td></td>
<td>@id(1) @optional string locale;</td>
</tr>
<tr>
<td></td>
<td>@id(2) @optional string text;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td>ExtensionObject</td>
<td>enum BodyEncoding {</td>
</tr>
<tr>
<td></td>
<td>@value(0) NONE_BODY_ENCODING,</td>
</tr>
<tr>
<td></td>
<td>@value(1) BYTESTRING_BODY_ENCODING,</td>
</tr>
<tr>
<td></td>
<td>@value(2) XMLELEMENT_BODY_ENCODING</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>@nested union ExtensionObjectBody switch (BodyEncoding) {</td>
</tr>
<tr>
<td></td>
<td>case NONE_BODY_ENCODING:</td>
</tr>
<tr>
<td></td>
<td>octet none_encoding;</td>
</tr>
<tr>
<td></td>
<td>case BYTESTRING_BODY_ENCODING:</td>
</tr>
<tr>
<td></td>
<td>sequence&lt;octet&gt; bytestring_encoding;</td>
</tr>
<tr>
<td></td>
<td>case XMLELEMENT_BODY_ENCODING:</td>
</tr>
<tr>
<td></td>
<td>XmlElement xmlelement_encoding;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>struct ExtensionObject {</td>
</tr>
<tr>
<td></td>
<td>NodeId type_id;</td>
</tr>
<tr>
<td></td>
<td>ExtensionObjectBody body;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td>DataValue</td>
<td>@mutable struct DataValue {</td>
</tr>
<tr>
<td></td>
<td>@id(1) @optional Variant value;</td>
</tr>
<tr>
<td></td>
<td>@id(2) @optional StatusCode status;</td>
</tr>
<tr>
<td></td>
<td>@id(4) @optional DateTime source_timestamp;</td>
</tr>
<tr>
<td></td>
<td>@id(8) @optional DateTime server_timestamp;</td>
</tr>
<tr>
<td></td>
<td>@id(10) @optional uint16 source_pico_sec;</td>
</tr>
<tr>
<td></td>
<td>@id(32) @optional uint16 server_pico_sec;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td>Variant</td>
<td>@nested union VariantValue switch (BuiltinTypeKind) {</td>
</tr>
<tr>
<td></td>
<td>case BOOLEAN_TYPE:</td>
</tr>
<tr>
<td></td>
<td>boolean bool_value;</td>
</tr>
<tr>
<td></td>
<td>case SBYTE_TYPE:</td>
</tr>
<tr>
<td></td>
<td>int8 sbyte_value;</td>
</tr>
<tr>
<td></td>
<td>case BYTE_TYPE:</td>
</tr>
<tr>
<td></td>
<td>uint8 byte_value;</td>
</tr>
<tr>
<td></td>
<td>case INT16_TYPE:</td>
</tr>
<tr>
<td></td>
<td>int16 int16_value;</td>
</tr>
<tr>
<td></td>
<td>case UINT16_TYPE:</td>
</tr>
<tr>
<td></td>
<td>uint16 uint16_value;</td>
</tr>
<tr>
<td></td>
<td>case INT32_TYPE:</td>
</tr>
<tr>
<td></td>
<td>int32 int32_value;</td>
</tr>
</tbody>
</table>
In the IDL representation of a `Variant`, `array_dimensions` may be set to a zero-length sequence, a sequence of length one, or a sequence of length greater than one:

- If `array_dimensions` is an empty zero-length sequence, it indicates the `Variant` contains a single element. In this case the value field shall contain a sequence of length 1 with that one element representing the value of the variant.
- If `array_dimensions` is a sequence of length 1, it indicates the `Variant` contains a one-dimensional array. In this case the first and only array_dimensions element shall match the length of the value sequence.
If `array_dimensions` is a sequence with length greater than 1, it indicates the Variant contains a multi-dimensional array. The length of `array_dimensions` indicates the number of dimensions and the value of each element in `array_dimensions` indicates the length of each dimension. As specified in [OPCUA-06], multi-dimensional arrays are encoded as a one-dimensional array whose length is equal to the sum of the lengths of each dimension with the higher rank dimensions appearing first. In this case, the `value` field shall contain a sequence with length equaling the sum of all the dimensions.

### 8.3 OPC UA Service Sets Mapping

This clause defines a set of DDS Services equivalent to the OPC UA Services specified in [OPCUA-04]. These allow DDS applications to browse, query, read, write, and subscribe to information in the AddressSpace of different OPC UA Servers in a pure client-server manner.

The DDS Services specified in this clause are built upon the mechanisms defined in [DDS-RPC] and [IDL], which provide IDL syntax to define interfaces with methods/operations and attributes, and the mapping of OPC UA’s built-in types specified in sub clause 8.2 of this specification.

Each DDS Service contains a group of methods with input and output parameter, which are identified with the `in` and `out` keywords (e.g., `out sequence<DataValue> results`). The first parameter of each method is always the input parameter `server_id`—a string that uniquely identifies the OPC UA Server that shall process the request. The format of the `server_id` is unspecified; it may be the Server’s URI (e.g., opc.tcp://10.10.100.131:55001) or an identifier corresponding a custom name specified in a configuration file. Aside from the output parameters, each method returns a `ResponseHeader`, whose mapping is specified in Table 8.3.

The standard DataTypes, NodeClasses, and Services mapped in this clause are also available in a separate machine-readable IDL file named `dds-opcua_services.idl`, which is provided with this specification.

#### 8.3.1 Standard DataTypes and NodeClasses Mapping

Table 8.3 maps the OPC UA DataTypes and NodeClasses that are required to implement DDS Services equivalent to those in [OPCUA-04].

These mappings are built upon the type mappings specified in sub clause 8.2 of this specification.

<table>
<thead>
<tr>
<th>OPC UA Type</th>
<th>DDS Type (IDL equivalent)³</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeClass</td>
<td>enum NodeClass {</td>
</tr>
<tr>
<td></td>
<td>@value(1) OBJECT_NODE_CLASS,</td>
</tr>
<tr>
<td></td>
<td>@value(2) VARIABLE_NODE_CLASS,</td>
</tr>
<tr>
<td></td>
<td>@value(4) METHOD_NODE_CLASS,</td>
</tr>
<tr>
<td></td>
<td>@value(8) OBJECT_TYPE_NODE_CLASS,</td>
</tr>
<tr>
<td></td>
<td>@value(16) VARIABLE_TYPE_NODE_CLASS,</td>
</tr>
<tr>
<td></td>
<td>@value(32) REFERENCE_TYPE_NODE_CLASS,</td>
</tr>
<tr>
<td></td>
<td>@value(64) DATA_TYPE_NODE_CLASS,</td>
</tr>
<tr>
<td></td>
<td>@value(128) VIEW_NODE_CLASS</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

³ All these types appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS`. 
<table>
<thead>
<tr>
<th>OPC UA Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
</table>
| BaseNodeClass       | @nested
|                     | struct BaseNodeClass {     |
|                     |   // Attributes           |
|                     |     NodeId node_id;       |
|                     |     NodeClass node_class;|
|                     |     QualifiedName browse_name;|
|                     |     LocalizedText display_name;|
|                     |     @optional LocalizedText description;|
|                     |     @optional uint32 write_mask;|
|                     |     @optional uint32 user_write_mask;|
|                     |   // No References specified for the BaseNodeClass |
| EnumValueType       | struct EnumValueType {    |
|                     |   int64 value;            |
|                     |   LocalizedText display_name;|
|                     |   LocalizedText description;|
| DataType            | @nested
|                     | struct DataType : BaseNodeClass {|
|                     |   // Attributes          |
|                     |     boolean is_abstract;|
|                     |   // References          |
|                     |     sequence<NodeId> has_property;|
|                     |     sequence<NodeId> has_subtype;|
|                     |     sequence<NodeId> has_encoding;|
|                     |   // Standard Properties |
|                     |     @optional string node_version;|
|                     |     @optional sequence<LocalizedText> enum_strings;|
|                     |     @optional sequence<EnumValueType> enum_values;|
|                     |     @optional sequence<LocalizedText> option_set_values;|
| BaseDataType        | Variant                   |
| Duration            | double                    |
| UtcTime             | DateTime                  |
| ContinuationPoint   | ByteString                 |
| Index               | uint32                    |
| IntegerId           | uint32                    |
| Counter             | uint32                    |
| NumericRange        | string                    |
| ViewDescription     | @nested
|                     | struct ViewDescription {  |
|                     |   NodeId view_id;         |
|                     |   UtcTime timestamp;      |
|                     |   uint32 view_version;    |
|                     | };                        |
| RelativePath        | @nested
<p>|                     | struct RelativePathElement {|
|                     |   NodeId reference_type_id;|
|                     |   boolean is_inverse;     |
|                     |   boolean include_subtypes;|
|                     |   QualifiedName target_name;|
|                     | };                        |</p>
<table>
<thead>
<tr>
<th>OPC UA Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelativePath</td>
<td><code>struct RelativePath {</code>&lt;br&gt;  <code>  sequence&lt;RelativePathElement&gt; elements;</code>&lt;br&gt;  <code>};</code></td>
</tr>
<tr>
<td>ReferenceDescription</td>
<td><code>struct ReferenceDescription {</code>&lt;br&gt;  <code>  NodeId reference_type_id;</code>&lt;br&gt;  <code>  boolean is_forward;</code>&lt;br&gt;  <code>  ExpandedNodeId node_id;</code>&lt;br&gt;  <code>  QualifiedName browse_name;</code>&lt;br&gt;  <code>  LocalizedText display_name;</code>&lt;br&gt;  <code>  NodeClass node_class;</code>&lt;br&gt;  <code>  ExpandedNodeId type_definition;</code>&lt;br&gt;  <code>};</code></td>
</tr>
<tr>
<td>BrowseResult</td>
<td><code>struct BrowseResult {</code>&lt;br&gt;  <code>  StatusCode status_code;</code>&lt;br&gt;  <code>  ContinuationPoint continuation_point;</code>&lt;br&gt;  <code>  sequence&lt;ReferenceDescription&gt; references;</code>&lt;br&gt;  <code>};</code></td>
</tr>
<tr>
<td>ResponseHeader</td>
<td><code>struct ResponseHeader {</code>&lt;br&gt;  <code>  UtcTime timestamp;</code>&lt;br&gt;  <code>  IntegerId request_handle;</code>&lt;br&gt;  <code>  StatusCode service_result;</code>&lt;br&gt;  <code>  DiagnosticInfo service_diagnostics;</code>&lt;br&gt;  <code>  sequence&lt;string&gt; string_table;</code>&lt;br&gt;  <code>};</code></td>
</tr>
<tr>
<td>ExtensibleParameter</td>
<td><code>struct ExtensibleParameter {</code>&lt;br&gt;  <code>  NodeId parameter_type_id;</code>&lt;br&gt;  <code>};</code></td>
</tr>
<tr>
<td>FilterOperator</td>
<td><code>enum FilterOperator {</code>&lt;br&gt;  <code>  @value(0) EQUALS_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(1) IS_NULL_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(2) GREATER_THAN_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(3) LESS_THAN_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(4) GREATER_THAN_OR_EQUAL_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(5) LESS_THAN_OR_EQUAL_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(6) LIKE_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(7) NOT_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(8) BETWEEN_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(9) IN_LIST_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(10) AND_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(11) OR_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(12) CAST_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(13) IN_VIEW_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(14) OF_TYPE_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(15) RELATED_TO_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(16) BITWISE_AND_FILTER_OPERATOR,</code>&lt;br&gt;  <code>  @value(17) BITWISE_OR_FILTER_OPERATOR</code>&lt;br&gt;  <code>};</code></td>
</tr>
<tr>
<td>FilterOperandKind</td>
<td><code>enum FilterOperandKind {</code>&lt;br&gt;  <code>  ELEMENT_FILTER_OPERAND_KIND,</code>&lt;br&gt;  <code>  LITERAL_FILTER_OPERAND_KIND,</code>&lt;br&gt;  <code>  ATTRIBUTE_FILTER_OPERAND_KIND</code>&lt;br&gt;  <code>};</code></td>
</tr>
<tr>
<td>OPC UA Type</td>
<td>DDS Type (IDL equivalent)</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>SIMPLE_ATTRIBUTE_FILTER_OPERAND_KIND</td>
<td></td>
</tr>
</tbody>
</table>
| @nested structural ElementOperand {
  uint32 index;
}; |
| @nested structural LiteralOperand {
  BaseDataType value;
}; |
| @nested structural AttributeOperand {
  NodeId node_id;
  string operand_alias;
  RelativePath browse_path;
  IntegerId attribute_id;
  NumericRange index_range;
}; |
| @nested structural SimpleAttributeOperand {
  NodeId type_id;
  sequence<QualifiedName> browse_path;
  IntegerId attribute_id;
  NumericRange index_range;
}; |
| @nested union FilterOperand switch (FilterOperandKind) {
  case ELEMENT_FILTER_OPERAND_KIND:
    ElementOperand element_operand;
  case LITERAL_FILTER_OPERAND_KIND:
    LiteralOperand literal_operand;
  case ATTRIBUTE_FILTER_OPERAND_KIND:
    AttributeOperand attribute_operand;
  case SIMPLE_ATTRIBUTE_FILTER_OPERAND_KIND:
    SimpleAttributeOperand simple_attribute_operand;
}; |
| structural ExtensibleParameterFilterOperand : ExtensibleParameter {
  FilterOperand parameter_data;
}; |
| @nested structural ContentFilterElement {
  FilterOperator filter_operator;
  sequence<ExtensibleParameterFilterOperand> filter_operands;
}; |
| @nested structural ContentFilterElementResult {
  StatusCode status_code;
  sequence<StatusCode> operand_status_codes;
  sequence<DiagnosticInfo> operand_diagnostic_infos;
}; |
<table>
<thead>
<tr>
<th>OPC UA Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
</table>
| struct ContentFilter {  
  sequence<ContentFilterElement> content_filter_element;  
};  
@nested  
struct ContentFilterResult {  
  sequence<ContentFilterElementResult> element_results;  
  sequence<DiagnosticInfo> element_diagnostic_infos;  
};  
| QueryDataSet | @nested  
struct QueryDataSet {  
  ExpandedNodeId node_id;  
  ExpandedNodeId type_definition_node;  
  sequence<BaseDataType> values;  
};  
| TimestampsToReturn | enum TimestampsToReturn {  
  @value(0) SOURCE_TIMESTAMPS_TO_RETURN,  
  @value(1) SERVER_TIMESTAMPS_TO_RETURN,  
  @value(2) BOTH_TIMESTAMPS_TO_RETURN,  
  @value(3) NEITHER_TIMESTAMPS_TO_RETURN  
};  
| ReadValueId | @nested  
struct ReadValueId {  
  NodeId node_id;  
  IntegerId attribute_id;  
  NumericRange index_range;  
  QualifiedName data_encoding;  
};  
| NotificationData | enum NotificationKind {  
  DATA_CHANGE_NOTIFICATION_KIND,  
  EVENT_NOTIFICATION_KIND,  
  STATUS_CHANGE_NOTIFICATION_KIND  
};  
@nested  
struct MonitoredItemNotification {  
  IntegerId client_handle;  
  DataValue value;  
};  
@nested  
struct DataChangeNotification {  
  sequence<MonitoredItemNotification> monitored_items;  
  sequence<DiagnosticInfo> diagnostic_infos;  
};  
@nested  
struct EventFieldList {  
  IntegerId client_handle;  
  sequence<BaseDataType> event_fields;  
};  
@nested  
struct EventNotificationList {  
  sequence<EventFieldList> events;  
};  
struct StatusChangeNotification {  

<table>
<thead>
<tr>
<th>OPC UA Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatusCode status; DiagnosticInfo diagnostic_info;</td>
<td></td>
</tr>
<tr>
<td>@nested union NotificationData switch(NotificationKind) { case DATA_CHANGE_NOTIFICATION_KIND: DataChangeNotification data_change_notification; case EVENT_NOTIFICATION_KIND: EventNotificationList event_notification_list; case STATUS_CHANGE_NOTIFICATION_KIND: StatusChangeNotification status_change_notification; }</td>
<td></td>
</tr>
<tr>
<td>@nested struct ExtensibleParameterNotificationData : ExtensibleParameter { NotificationData parameter_data; }</td>
<td></td>
</tr>
<tr>
<td>@nested struct NotificationMessage { Counter sequence_number; UtcTime publish_time; sequence&lt;ExtensibleParameterNotificationData&gt; notification_data; }</td>
<td></td>
</tr>
<tr>
<td>MonitoringFilter Parameters</td>
<td></td>
</tr>
<tr>
<td>enum MonitoringFilterKind { DATA_CHANGE_MONITORING_FILTER_KIND, EVENT_MONITORING_FILTER_KIND, AGGREGATE_MONITORING_FILTER_KIND }</td>
<td></td>
</tr>
<tr>
<td>enum DataChangeTrigger { @value(0) STATUS_DATA_CHANGE_TRIGGER, @value(1) STATUS_VALUE_DATA_CHANGE_TRIGGER, @value(2) STATUS_VALUE_TIMESTAMP_DATA_CHANGE_TRIGGER }</td>
<td></td>
</tr>
<tr>
<td>@nested struct DataChangeFilter { DataChangeTrigger trigger; uint32 deadband_type; double deadband_value; }</td>
<td></td>
</tr>
<tr>
<td>@nested struct EventFilter { sequence&lt;SimpleAttributeOperand&gt; select_clauses; ContentFilter where_clause; }</td>
<td></td>
</tr>
<tr>
<td>@nested struct AggregateConfiguration { boolean user_server_capabilities_defaults; boolean treat_uncertain_as_bad; octet percent_data_bad; octet percent_data_good; boolean use_sloped_extrapolation; }</td>
<td></td>
</tr>
</tbody>
</table>
### OPC UA Type

<table>
<thead>
<tr>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>@nested</td>
</tr>
<tr>
<td>struct AggregateFilter {</td>
</tr>
<tr>
<td>UtcTime start_time;</td>
</tr>
<tr>
<td>NodeId aggregate_type;</td>
</tr>
<tr>
<td>Duration processing_interval;</td>
</tr>
<tr>
<td>AggregateConfiguration aggregate_configuration;</td>
</tr>
<tr>
<td>};</td>
</tr>
<tr>
<td>@nested</td>
</tr>
<tr>
<td>union MonitoringFilter switch (MonitoringFilterKind) {</td>
</tr>
<tr>
<td>case DATA_CHANGE_MONITORING_FILTER_KIND:</td>
</tr>
<tr>
<td>DataChangeFilter data_change_filter;</td>
</tr>
<tr>
<td>case EVENT_MONITORING_FILTER_KIND:</td>
</tr>
<tr>
<td>EventFilter event_filter;</td>
</tr>
<tr>
<td>case AGGREGATE_MONITORING_FILTER_KIND:</td>
</tr>
<tr>
<td>AggregateFilter aggregate_filter_result;</td>
</tr>
<tr>
<td>};</td>
</tr>
<tr>
<td>@nested</td>
</tr>
<tr>
<td>struct ExtensibleParameterMonitoringFilter : ExtensibleParameter</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>MonitoringFilter parameter_data;</td>
</tr>
<tr>
<td>};</td>
</tr>
<tr>
<td>@nested</td>
</tr>
<tr>
<td>struct EventFilterResult {</td>
</tr>
<tr>
<td>sequence&lt;StatusCode&gt; select_clause_results;</td>
</tr>
<tr>
<td>sequence&lt;DiagnosticInfo&gt; select_clause_diagnostic_infos;</td>
</tr>
<tr>
<td>ContentFilterResult where_clause_result;</td>
</tr>
<tr>
<td>};</td>
</tr>
<tr>
<td>@nested</td>
</tr>
<tr>
<td>struct AggregateFilterResult {</td>
</tr>
<tr>
<td>UtcTime revised_start_time;</td>
</tr>
<tr>
<td>Duration revised_processing_interval;</td>
</tr>
<tr>
<td>};</td>
</tr>
<tr>
<td>@nested</td>
</tr>
<tr>
<td>union MonitoringFilterResult switch (MonitoringFilterKind) {</td>
</tr>
<tr>
<td>case EVENT_MONITORING_FILTER_KIND:</td>
</tr>
<tr>
<td>EventFilterResult event_filter_result;</td>
</tr>
<tr>
<td>case AGGREGATE_MONITORING_FILTER_KIND:</td>
</tr>
<tr>
<td>AggregateFilterResult aggregate_filter_result;</td>
</tr>
<tr>
<td>};</td>
</tr>
<tr>
<td>@nested</td>
</tr>
<tr>
<td>struct ExtensibleParameterMonitoringFilterResult</td>
</tr>
<tr>
<td>{ ExtensibleParameter {</td>
</tr>
<tr>
<td>MonitoringFilterResult parameter_data;</td>
</tr>
<tr>
<td>};</td>
</tr>
<tr>
<td>MonitoringMode</td>
</tr>
<tr>
<td>enum MonitoringMode {</td>
</tr>
<tr>
<td>@value(0) DISABLED_MONITORING_MODE,</td>
</tr>
<tr>
<td>@value(1) SAMPLING_MONITORING_MODE,</td>
</tr>
<tr>
<td>@value(2) REPORTING_MONITORING_MODE</td>
</tr>
<tr>
<td>};</td>
</tr>
<tr>
<td>MonitoringParameters</td>
</tr>
<tr>
<td>@nested</td>
</tr>
<tr>
<td>struct MonitoringParameters</td>
</tr>
<tr>
<td>{ IntegerId client_handle;</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
### 8.3.2 View Service Set

This sub clause defines an equivalent View Service Set using the DDS constructs defined in [DDS-RPC] for DDS applications that may want to navigate the AddressSpace of an OPC UA Server.

#### 8.3.2.1 Type Definitions

Table 8.4 shows the mapping of the types specific to the View Service Set. All these types appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS::VIEW`.

<table>
<thead>
<tr>
<th>OPC UA Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowsePath</td>
<td><code>struct BrowsePath { NodeId starting_node; RelativePath relative_path; };</code></td>
</tr>
<tr>
<td>BrowsePathResult</td>
<td><code>struct BrowsePathTarget { ExpandedNodeId target_id; Index remaining_path_index; };</code></td>
</tr>
<tr>
<td></td>
<td><code>struct BrowsePathResult { StatusCode status_code; sequence&lt;BrowsePathTarget&gt; targets; };</code></td>
</tr>
<tr>
<td>BrowseDirection</td>
<td><code>enum BrowseDirection { @value(0) FORWARD_BROWSE_DIRECTION, @value(1) REVERSE_BROWSE_DIRECTION, @value(3) BOTH_BROWSE_DIRECTION };</code></td>
</tr>
<tr>
<td>BrowseDescription</td>
<td><code>struct BrowseDescription { NodeId node_id; BrowseDirection browse_direction; NodeId reference_type_id; boolean include_subtypes; uint32 node_class_mask; uint32 result_mask; };</code></td>
</tr>
</tbody>
</table>
8.3.2.2 Service Interfaces

The following IDL defines the interfaces to be implemented by the DDS View Service Set using the syntax defined in [DDS-RPC] and [IDL].

The Service and all its methods appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS::VIEW`.

```idl
@DDSService
interface View {
    ResponseHeader browse(
        string server_id, // Identifies OPC UA server
        out sequence<BrowseResult> results,
        out sequence<DiagnosticInfo> diagnostic_infos,
        in ViewDescription view_description,
        in Counter requested_max_references_per_node,
        in sequence<BrowseDescription> nodes_to_browse);

    ResponseHeader browse_next(
        string server_id, // Identifies OPC UA server
        out sequence<BrowseResult> results,
        out sequence<DiagnosticInfo> diagnostic_infos,
        in boolean release_continuation_points,
        in sequence<ContinuationPoint> continuation_points);

    ResponseHeader translate_browse_paths_to_node_ids(
        string server_id, // Identifies OPC UA server
        out sequence<BrowsePathResult> results,
        out sequence<DiagnosticInfo> diagnostic_infos,
        in sequence<BrowsePath> browse_paths);

    ResponseHeader register_nodes(
        string server_id, // Identifies OPC UA server
        out sequence<NodeId> registered_node_ids,
        in sequence<NodeId> nodes_to_register);

    ResponseHeader unregister_nodes(
        string server_id, // Identifies OPC UA server
        in sequence<NodeId> nodes_to_unregister)
};
```

8.3.3 Query Service Set

This sub clause defines an equivalent Query Service Set using the DDS constructs defined in [DDS-RPC] for DDS applications that may obtain information from the AddressSpace of an OPC UA Server.

8.3.3.1 Type Definitions

Table 8.5 shows the mapping of the types specific to the Query Service Set. All these types appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS::QUERY`.

<table>
<thead>
<tr>
<th>OPC UA Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParsingResult</td>
<td>@nested</td>
</tr>
</tbody>
</table>

Table 8.5: Mapping of Types Specific to the Query Service Set
8.3.3.2 Service Interfaces

The following IDL defines the interfaces to be implemented by the DDS Query Service Set using the syntax defined in [DDS-RPC] and [IDL].

The Service and all its methods appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS::QUERY`.

```idl
@DDSService
interface Query {
    ResponseHeader query_first(
        string server_id, // Identifies OPC UA server
        out sequence<QueryDataSet> query_data_sets,
        out ContinuationPoint continuation_point,
        out sequence<ParsingResult> parsing_results,
        out sequence<DiagnosticInfo> diagnostic_infos,
        out ContentFilterResult filter_result,
        in ViewDescription view,
        in sequence<NodeTypeDescription> node_types,
        in ContentFilter filter,
        in Counter max_datasets_to_return,
        in Counter max_references_to_return);

    ResponseHeader query_next(
        string server_id, // Identifies OPC UA server
        out sequence<QueryDataSet> query_data_sets,
        out ContinuationPoint revised_continuation_point,
        in boolean release_continuation_point,
        in ContinuationPoint continuation_point);
};
```

8.3.4 Attribute Service Set

This sub clause defines an equivalent Attribute Service Set using the DDS constructs defined in [DDS-RPC] for DDS applications that may want to perform read or write operations (and their equivalent for historical data) on Attributes from Nodes in the AddressSpace of an OPC UA Server.
8.3.4.1 Type Definitions

Table 8.6 shows the mapping of the types specific to the Attribute Service Set. All these types appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS::ATTRIBUTE`.

Table 8.6: Mapping of Types Specific to the Attribute Service Set

<table>
<thead>
<tr>
<th>OPC UA Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
</table>
| HistoryData Parameters            | @nested struct HistoryEventFieldList {
|                                   |   sequence<BaseDataType> event_fields;
|                                   | };
|                                   | @nested struct HistoryEvent {
|                                   |   sequence<HistoryEventFieldList> events;
|                                   | };
|                                   | @nested struct HistoryData {
|                                   |   sequence<DataValue> data_values;
|                                   | };
|                                   | @nested struct ExtensibleParameterHistoryData : ExtensibleParameter {
|                                   |   HistoryData parameter_data;
|                                   | };
| HistoryReadResult                 | @nested struct HistoryReadResult {
|                                   |   StatusCode status_code;
|                                   |   ContinuationPoint continuation_point;
|                                   |   ExtensibleParameterHistoryData history_data;
|                                   | };
| HistoryReadValueId                | @nested struct HistoryReadValueId {
|                                   |   NodeId node_id;
|                                   |   NumericRange index_range;
|                                   |   QualifiedName data_encoding;
|                                   |   ContinuationPoint continuation_point;
|                                   | };
| WriteValue                        | @nested struct WriteValue {
|                                   |   NodeId node_id;
|                                   |   IntegerId attribute_id;
|                                   |   NumericRange index_range;
|                                   |   DataValue value;
|                                   | };
| HistoryUpdateResult               | @nested struct HistoryUpdateResult {
|                                   |   StatusCode status_code;
|                                   |   sequence<StatusCode> operation_results;
|                                   |   sequence<DiagnosticInfo> diagnostic_infos;
|                                   | };
| HistoryUpdateType                 | enum HistoryUpdateType {
|                                   |   @value(1) INSERT_HISTORY_UPDATE_TYPE,
|                                   |   @value(2) REPLACE_HISTORY_UPDATE_TYPE,
|                                   | };

4 Some of the types defined are part of [OPCUA-11], which focuses on the Historical Access functionality of the OPC UA standard.
@value(3) UPDATE_HISTORY_UPDATE_TYPE,
@value(4) DELETE_HISTORY_UPDATE_TYPE
};

@nested
struct ExtensibleParameterHistoryUpdate : ExtensibleParameter {
    HistoryUpdateType parameter_data;
};

enum HistoryReadDetailsKind {
    READ_EVENT_HISTORY_READ_DETAILS_KIND,
    READ_RAW_MODIFIED_HISTORY_READ_DETAILS_KIND,
    READ_PROCESSED_HISTORY_READ_DETAILS_KIND,
    READ_AT_TIME_HISTORY_READ_DETAILS_KIND
};

@nested
struct ReadEventDetails {
    Counter num_values_per_node;
    UtcTime start_time;
    UtcTime end_time;
    EventFilter filter;
};

@nested
struct ReadRawModifiedDetails {
    boolean is_read_modified;
    UtcTime start_time;
    UtcTime end_time;
    Counter num_values_per_node;
    boolean return_bounds;
};

struct ReadProcessedDetails {
    UtcTime start_time;
    UtcTime end_time;
    Duration processing_interval;
    sequence<NodeId> aggregate_type;
    AggregateConfiguration aggregate_configuration;
};

struct ReadAtTimeDetails {
    sequence<UtcTime> req_times;
    boolean use_simple_bounds;
};

@nested
union HistoryReadDetails switch (HistoryReadDetailsKind) {
    case READ_EVENT_HISTORY_READ_DETAILS_KIND:
        ReadEventDetails read_event_details;
    case READ_RAW_MODIFIED_HISTORY_READ_DETAILS_KIND:
        ReadRawModifiedDetails read_raw_modified_details;
    case READ_PROCESSED_HISTORY_READ_DETAILS_KIND:
        ReadProcessedDetails read_processed_details;
    case READ_AT_TIME_HISTORY_READ_DETAILS_KIND:
        ReadAtTimeDetails read_at_time_details;
};

@nested
struct ExtensibleParameterHistoryReadDetails : ExtensibleParameter {
### OPC UA Type

<table>
<thead>
<tr>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HistoryReadDetails parameter_data;</td>
</tr>
</tbody>
</table>

#### PerformUpdateType

<table>
<thead>
<tr>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum PerformUpdateType { @value(1) INSERT_PERFORM_UPDATE_TYPE, @value(2) REPLACE_PERFORM_UPDATE_TYPE, @value(3) UPDATE_PERFORM_UPDATE_TYPE, @value(4) REMOVE_PERFORM_UPDATE_TYPE</td>
</tr>
</tbody>
</table>

#### HistoryUpdateDetails

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
</table>
| struct UpdateDataDetails {
| NodeId node_id;
| PerformUpdateType perform_insert_replace;
| sequence<DataValue> update_values;
| } |

<table>
<thead>
<tr>
<th>@nested</th>
</tr>
</thead>
</table>
| struct UpdateStructureDataDetails {
| NodeId node_id;
| PerformUpdateType perform_insert_replace;
| sequence<DataValue> update_values;
| } |

<table>
<thead>
<tr>
<th>@nested</th>
</tr>
</thead>
</table>
| struct UpdateEventDetails {
| NodeId node_id;
| PerformUpdateType perform_insert_replace;
| EventFilter filter;
| sequence<HistoryEventFieldList> event_data;
| } |

<table>
<thead>
<tr>
<th>@nested</th>
</tr>
</thead>
</table>
| struct DeleteRawModifiedDetails {
| NodeId node_id;
| boolean is_delete_modified;
| UtcTime start_time;
| UtcTime end_time;
| } |

<table>
<thead>
<tr>
<th>@nested</th>
</tr>
</thead>
</table>
| struct DeleteAtTimeDetails {
| NodeId node_id;
| sequence<UtcTime> req_times;
| } |

<table>
<thead>
<tr>
<th>@nested</th>
</tr>
</thead>
</table>
| struct DeleteEventDetails {
| NodeId node_id;
| sequence<ByteString> event_id;
| } |

#### HistoryUpdateDetailsKind

<table>
<thead>
<tr>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
</table>
| enum HistoryUpdateDetailsKind {
| UPDATE_DATA_HISTORY_UPDATE_DETAILS_KIND, UPDATE_STRUCTURE_HISTORY_UPDATE_DETAILS_KIND, UPDATE_EVENT_HISTORY_UPDATE_DETAILS_KIND, DELETE_RAW_MODIFIED_HISTORY_UPDATE_DETAILS_KIND, DELETE_AT_TIMES_HISTORY_UPDATE_DETAILS_KIND, DELETE_EVENTS_HISTORY_UPDATE_DETAILS_KIND |
| } |
8.3.4.2 Service Interfaces

The following IDL defines the interfaces to be implemented by the DDS Attribute Service Set using the syntax defined in [DDS-RPC] and [IDL].

The Service and all its methods appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS::ATTRIBUTE`.

```idl
@DDSService
interface Attribute {
    ResponseHeader read(
        string server_id, // Identifies OPC UA server
        out sequence<DataValue> results,
        out sequence<DiagnosticInfo> diagnostic_infos,
        in Duration max_age,
        in TimestampsToReturn timestamps_to_return,
        in sequence<ReadValueId> nodes_to_read);

    ResponseHeader history_read(
        string server_id, // Identifies OPC UA server
        out sequence<HistoryReadResult> results,
        out sequence<DiagnosticInfo> diagnostic_infos,
        in ExtensibleParameterHistoryReadDetails history_read_details,
        in TimestampsToReturn timestamps_to_return,
        in boolean release_continuation_points,
        in sequence<HistoryReadValueId> nodes_to_read);

    ResponseHeader write(
        string server_id, // Identifies OPC UA server
        out sequence<StatusCode> results,
        out sequence<DiagnosticInfo> diagnostic_infos,
        in sequence<WriteValue> nodes_to_write);

    ResponseHeader history_update(
        string server_id, // Identifies OPC UA server
        out sequence<HistoryUpdateResult> results,
        out sequence<DiagnosticInfo> diagnostic_infos,
```
8.3.5 Method Service Set

This sub clause defines an equivalent Method Service Set using the DDS constructs defined in [DDS-RPC] for DDS applications that may want to invoke methods available in the AddressSpace of an OPC UA Server.

8.3.5.1 Type Definitions

Table 8.7 shows the mapping of the types specific to the Method Service Set. All these types appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS::METHOD`.

<table>
<thead>
<tr>
<th>OPC UA Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
</table>
| CallMethodRequest    | @nested struct CallMethodRequest {
|                      |   NodeId object_id;
|                      |   NodeId method_id;
|                      |   sequence<BaseDataType> input_arguments;
|                      | };                       |
| CallMethodResult     | @nested struct CallMethodResult {
|                      |   StatusCode status_code;
|                      |   sequence<StatusCode> input_arguments_results;
|                      |   sequence<DiagnosticInfo> input_arguments_diagnostic_infos;
|                      |   sequence<BaseDataType> output_arguments;
|                      | };                       |

8.3.5.2 Service Interfaces

The following IDL defines the interfaces to be implemented by the DDS Method Service Set using the syntax defined in [DDS-RPC] and [IDL].

The Service and all its methods appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS::METHOD`.

```idl
@DDSService
interface Method {
   ResponseHeader call(
      string server_id, // Identifies OPC UA server
      out sequence<CallMethodResult> results,
      out sequence<DiagnosticInfo> diagnostic_infos,
      in sequence<CallMethodRequest> methods_to_call);
}
```

8.3.6 Implementation Considerations

8.3.6.1 OPC UA Implementation Considerations

The representation of the OPC UA Service Sets using RPC over DDS specified in this chapter requires the OPC UA/DDS Gateway to embed one or more OPC UA Clients. These OPC UA Clients shall be capable of:

- Connecting to OPC UA Servers using the Discovery, SecureChannel, and Session Service Sets.
- Browsing the AddressSpace of OPC UA Servers using the View Service Set.
• Obtaining information from the AddressSpace of Servers using the Query Service Set.
• Reading and Writing Attributes using the Attribute Service Set.
• Calling Methods on OPC UA Servers using the Method Service Set.

To comply with all the requirements listed above, implementers of this specification shall use an OPC UA Client compliant with the Standard UA Client Profile defined in sub clause 6.5.121 of [OPCUA-07]. Alternatively, implementers of this specification may use an OPC UA Client that is not fully compliant with the Standard UA Client Profile, but complies with the following Client Facets specified in [OPCUA-07]:

• Core Client Facet
• Base Client Behavior Facet
• Discovery Client Facet
• AddressSpace Lookup Client Facet
• Attribute Read Client Facet
• Attribute Write Client Facet
• Method Client Facet

Additionally, OPC UA Clients (whether they are compliant with Standard UA Client Profile or compliant with the required Client Facets listed above) shall support an extra facet to access historical data: the Historical Access Client Facet defined in sub clause 6.5.97 of [OPCUA-07].

Consequently, compliant implementations of this specification shall be built upon an OPC UA implementation capable of passing the conformance tests specified for those profiles and facets by the OPC Foundation.

Lastly, it is important to note that implementers of this specification may need to configure the underlying OPC UA Clients—which provide access to the mapped Services—to satisfy the requirements of remote OPC UA Servers in terms of authentication, access control, and encryption using the mechanisms provided by the OPC UA Security Model [OPCUA-02]. Depending on the requirements of the remote OPC UA Servers, OPC UA Clients may need to support additional security-related facets from [OPCUA-07].

### 8.3.6.2 DDS Implementation Considerations

To implement the mappings specified in this chapter OPC UA/DDS Gateway shall use a DDS implementation complaint with:

• Minimum Profile of [DDS].
• Statements listed in clause 8.4.2 of [DDSI-RTPS].
• Basic Conformance Profile of [DDS-RPC].
• Minimal Conformance Profile of [DDS-XTYPES].

Some deployments may require the mechanisms specified in [DDS-SECURITY] to enable the DDS side of the OPC UA/DDS Gateway to access secured Domains and Topics for publishing and subscribing to information. In those cases, the underlying DDS implementation shall also be compliant with the Built-in Plugin Interoperability and Plugin Framework Conformance Points of [DDS-SECURITY].

As specified in the rest of clauses dealing with DDS and OPC UA integration, the Gateway shall be capable of dealing with two different security models: the OPC UA Security Model on one end and the DDS Security Model on the other end. Each security model shall be configured separately depending on the needs of the end user of the OPC UA/DDS
Gateway. This specification does not directly address these aspects because they are fully described in [OPCUA-02] and [DDS-SECURITY].

8.4 OPC UA Subscription Model Mapping

8.4.1 Overview (non-normative)

As described in sub clause 7.1.2, the OPC UA Subscription and MonitoredItems Service Set provide Clients with a mechanism to receive Notifications from Servers on data changes and events.

This subscription model requires Client applications to connect to a Server, create a Session, configure a Subscription, associate a set of MonitoredItems, and send Publish requests to receive Notifications. Unlike in DDS, OPC UA Subscriptions are Client-specific and cannot be shared with other Clients.

8.4.1.1 Subscriptions

Subscriptions provide the channel through which Servers deliver Notifications to Clients. The Subscription Service Set is specified in clause 5.13 of [OPCUA-04].

To create a Subscription, Clients use the CreateSubscription service, which may be mapped to the following IDL:

```idl
ResponseHeader create_subscription(
    out IntegerId subscription_id,
    out Duration revised_publishing_interval,
    out Counter revised_lifetime_count,
    out Count revised_max_keep_alive_count,
    in Duration requested_publishing_interval,
    in Counter requested_lifetime_count,
    in Counter requested_max_keep_alive_count,
    in Counter max_notifications_per_publish,
    in boolean publishing_enabled,
    in octet priority);
```

Where:

- **subscription_id** is a numeric value that identifies the created Subscription.

- **requested_publishing_interval** is the rate at which the Subscription should deliver Notifications to the Client. The Server returns revised_publishing_interval—the negotiated value—as part of the response to the CreateSubscription request. If the requested value is 0 or negative, the Server will use the fastest supported publishing interval.

- **requested_lifetime_count** is the number of times the publishing timer may expire (without sending a NotificationMessage) before the Server closes the Subscription. It must be at least three times greater than the value of the RequestedMaxKeepAliveCount. The Server returns revised_lifetime_count—the negotiated value—as part of the response to the CreateSubscription request.

- **requested_max_keep_alive_count** is the number of times the publishing timer may expire (without sending a NotificationMessage) before the Subscription sends a keep-alive Message to the Client to ensure the Subscription remains in use. The Server returns revised_lifetime_count—the negotiated value—as part of the response to the CreateSubscription request. If the requested value is 0, the Server will use the smallest supported keep-alive count.

- **max_notifications_per_publish** is the maximum number of Notifications that the Client wants to receive in response to a single Publish request. If the requested value is zero, the Server will respond with all the Notifications queued to be sent.
- **publishing_enabled** indicates whether publishing is enabled for the Subscription.
- **priority** is the relative priority of the Subscription. The value is used to decide which of the competing Subscription sends Notifications as to respond a Publish request.

### 8.4.1.2 MonitoredItems

MonitoredItems identify the resources that a Client may monitor. To create a MonitoredItem—adding it to an existing Subscription—Clients use the CreateMonitoredItem service, which may be mapped to the following IDL:

```idl
ResponseHeader create_monitored_items(
    out sequence<MonitoredItemCreateResult> results,
    out sequence<DiagnosticInfo> diagnostic_infos,
    in IntegerId subscription_id,
    in TimestampsToReturn timestamps_to_return,
    in sequence<MonitoredItemCreateRequest> items_to_create);
```

Where:

- **subscription_id** is the numeric value that identifies the Subscription Notifications regarding the MonitoredItem will be sent through.
- **timestamps_to_return** specifies the timestamp attributes to be transmitted for each MonitoredItem.
- **items_to_create** contains a list with the MonitoredItems to be created as part of the CreateMonitoredItems request. Each MonitoredItemCreateRequest includes information to identify the MonitoredItem and the parameters that configure the sampling behavior (e.g., sampling interval, filters, queue size, etc.):
  ```idl
  @nested
  struct MonitoringParameters {
    IntegerId client_handle;
    Duration sampling_interval;
    ExtensibleParameterMonitoringFilter filter;
    Counter queue_size;
    boolean discard_oldest;
  }
  }
  @nested
  struct MonitoredItemCreateRequest {
    ReadValueId item_to_monitor;
    MonitoringMode monitoring_mode;
    MonitoringParameters monitoring_parameters;
  }
  ```
- **results** lists the result of the create operation in every MonitoredItem in items_to_create, this includes a status code, the assigned monitored_item_id, revised sampling interval, etc.
  ```idl
  @nested
  struct MonitoredItemCreateResult {
    StatusCode status_code;
    IntegerId monitored_item_id;
    Duration revised_sampling_interval;
    Counter revised_queue_size;
    ExtensibleParameterMonitoringFilterResult filter_result;
  }
  ```
- **diagnostic_infos** lists the diagnostic information for every MonitoredItem in items_to_create.

### 8.4.1.3 Notification Messages

NotificationMessages are sent to Client application as a response to Publish requests. Publish requests are queued at the Session level get dequeued by a Subscription in every publishing cycle. Therefore, Clients must issue enough Publish requests to the Server to guarantee the delivery of NotificationMessages.
NotificationMessages contain a sequence number that identifies them, a publication time, and a sequence of notification data. There are three kinds of NotificationMessages: DataChange, Event, and StatusChange.

### 8.4.1.3.1 DataChange Notifications

DataChange Notifications\(^5\) contain a sequence of MonitoredItems for which a change has been detected and a sequence of Diagnostic Information for each MonitoredItem. The equivalent IDL representation is specified in OPC UA Service Sets Mapping (Table 8.3):

```idl
@nested
struct MonitoredItemNotification {
    IntegerId client_handle;
    DataValue value;
};

@nested
struct DataChangeNotification {
    sequence<MonitoredItemNotification> monitored_items;
    sequence<DiagnosticInfo> diagnostic_infos;
};
```

The value of each MonitoredItem Notification is represented as a DataValue type, which contains the status code, value, and timestamp of the Attribute that is being monitored. The equivalent IDL representation is specified in OPC UA Service Sets Mapping (Table 8.3):

```idl
@mutable
struct DataValue {
    @id(1)  @optional Variant value;
    @id(2)  @optional StatusCode status;
    @id(4)  @optional DateTime source_timestamp;
    @id(8)  @optional DateTime server_timestamp;
    @id(10) @optional uint16 source_pico_sec;
    @id(32) @optional uint16 server_pico_sec;
};
```

To simplify the representation of MonitoredItems in DDS, this sub clause focuses only on the Value field of the MonitoredItems’ DataValue. Timestamps and status codes are therefore ignored.

The Value field of a DataValue is represented as Variant type, which provides a powerful mechanism to represent scalar values, arrays, and multi-dimensional for every OPC UA built-in type. OPC UA Type System Mapping defines in Table 8.2 a mapping of Variant to the DDS types system.

```idl
struct Variant {
    sequence<uint32> array_dimensions;
    sequence<VariantValue> value;
};
```

However, this direct mapping is difficult to handle for a typical DDS application, because it requires dealing with VariantValues, which are unions of all the OPC UA equivalent types; and array_dimensions, which represent the dimensions of the Variant—in other words, whether it is a scalar value, an array, or a multi-dimensional array.

### 8.4.1.3.2 Event Notifications

Event Notifications\(^6\) contain a sequence of Events that have been triggered. The equivalent IDL representation is specified in OPC UA Service Sets Mapping (Table 8.3) is the following:

```idl
@nested
struct EventFieldList {
    IntegerId client_handle;
    sequence<Variant> event_fields;
};
```

---

5. Data Change Notifications are specified in sub clause 7.20.2 of [OPCUA-04].

6. Event Notifications are specified in sub clause 7.20.3 of [OPCUA-04].
Each Event contains an array of one or more fields that describe it. The sequence of fields in each Event depends on both the type of Event and the EventFilter the MonitoredItem was created with. [OPCUA-03] lists thirty-four standard EventTypes, whose representation is specified in [OPCUA-05]. Alarms and Conditions, specified in [OPCUA-09], extend the Event handling to provide such functionality.

Every EventType inherits contains a common set of EventFields provided by the BaseEventType and may a group of Event-specific fields. The list of common EventFields is the following:

- EventId—Identifies a particular Event Notification.
- EventType—Describes the specific type of Event.
- SourceNode—Node that originated the Event.
- SourceName—Description of the source of the Event.
- Time—Provides the time the event occurred.
- ReceiveTime—Provides the time the OPC UA Server received the Event.
- LocalTime—Provides information on the offset between the Time property and the time at the location where the event was issued.
- Message—Localizable text description of the Event.
- Severity—Indicates the urgency of the Event, being 1 the lowest severity and 1,000 the highest.

Each EventField is represented as Variant, which—like in the case of Data Change Notifications—provides a mechanism to represent any kind of information.

### 8.4.1.3.3 StatusChange Notifications

StatusChange Notifications are used to report changes in the status of a Subscription.

### 8.4.2 OPC UA Subscription Mapping

This clause describes the simplified mapping of the OPC UA Subscription model to DDS. In particular, it specifies how to configure the OPC UA/DDS Gateway to create Subscriptions with Data and Event MonitoredItems, and how to map DataChange and Event NotificationMessages to DDS Topics.

#### 8.4.2.1 Overview

To map OPC UA Subscriptions and MonitoredItems to DDS Topics, the OPC UA/DDS Gateway introduces the concept of Subscription Mapping. This part of the OPC UA to DDS Bridge associates OPC UA Inputs (i.e., OPC UA Subscriptions) with DDS outputs (i.e., DDS Publications).

The relationship between OPC UA Inputs and a DDS Outputs is many-to-many: an OPC UA Input may be assigned to multiple DDS Outputs, and a DDS Output may be assigned values from multiple OPC UA Inputs.
Table 8.8 provides the IDL definition of Subscription Mapping Configuration.

Table 8.8: Subscription Mapping Configuration

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition (IDL Equivalent)</th>
</tr>
</thead>
</table>
| SubscriptionMapping   | struct SubscriptionMapping {
|                       |     sequence<OpcUaInput> opcua_inputs;
|                       |     sequence<DdsOutput> dds_outputs;
|                       |     InputOutputMapping mapping;
|                       | }                                                                |

8.4.2.2 OPC UA Inputs

The OPC UA/DDS Gateway may create Subscriptions to multiple OPC UA Servers using different OPC UA Clients embedded into the Gateway. Ideally, the Gateway should maintain a single Subscription with each monitored OPC UA Server to minimize the number of resources associated with the connection. However, because users may wish to define different Subscriptions to maintain—for instance—different publishing intervals for the same MonitoredItems, the Gateway shall allow the creation of more than one Subscription to the same OPC UA Server.
Table 8.9 shows the configuration of an OPC UA Input, which is comprised of two properties: `SubscriptionProtocol` and `MonitoredItems`.

### Table 8.9: OPC UA Input Definition

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition (IDL Equivalent)</th>
</tr>
</thead>
</table>
| OpcUaInput         | `@nested
struct OpcUaInput {
    string name;
    OpcUaConnection opcua_connection;
    SubscriptionProtocol subscription_protocol;
    sequence<MonitoredItem> monitored_items;
};` |

#### 8.4.2.2.1 Input Name

Every OPC UA Input is given a name that is necessary to identify the `MonitoredItems` associated with specific inputs in the mapping section.
8.4.2.2 OPC UA Connections

An OPC UA Connection configuration provides all the necessary information for the OPC UA Clients embedded into the Gateway to establish the connections that shall be used to create subscriptions on remote OPC UA Servers.

Table 8.10 provides the IDL definition of an OPC UA Connection and its connection settings.

**Table 8.10: OPC UA Connection Definition**

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition (IDL Equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpcUaConnection</td>
<td>@nested&lt;br&gt;struct OpcUaConnection {&lt;br&gt;    string endpoint_url;&lt;br&gt;    uint32 timeout;&lt;br&gt;    uint32 secure_channel_lifetime;&lt;br&gt;    OpcUaConectionConfig local_connection;&lt;br&gt;}</td>
</tr>
<tr>
<td>OpcUaConectionConfig</td>
<td>@nested&lt;br&gt;struct OpcUaConectionConfig {&lt;br&gt;    uint32 protocol_version;&lt;br&gt;    uint32 send_buffer_size;&lt;br&gt;    uint32 recv_buffer_size;&lt;br&gt;    uint32 max_message_size;&lt;br&gt;    uint32 max_chunk_count;&lt;br&gt;}</td>
</tr>
</tbody>
</table>

8.4.2.3 Subscription Protocol

Table 8.11 provides the IDL representation of the SubscriptionProtocol parameters. Each of these parameters is described in detail in sub clause 8.4.1.1.

**Table 8.11: OPC UA Subscription Protocol Definition**

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition (IDL Equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubscriptionProtocol</td>
<td>@nested&lt;br&gt;struct SubscriptionProtocol {&lt;br&gt;    double requested_publishing_interval;&lt;br&gt;    uint32 requested_lifetime_count;&lt;br&gt;    uint32 requested_max_keepalive_count;&lt;br&gt;    uint32 max_notifications_per_publish;&lt;br&gt;    boolean publishing_enabled;&lt;br&gt;    octet priority;&lt;br&gt;}</td>
</tr>
</tbody>
</table>

8.4.2.4 Monitored Items

MonitoredItemsList contains a collection of DataItems and EventItems, which represent Data Value and Event MonitoredItems, respectively.

Each DataItem is identified by a name and contains the following configuration parameters:

- **NodeId (NodeId)** as defined in Table 8.2—Identifies the Node containing the DataItem within the AddressSpace of an OPC UA Server.
- **AttributeId (uint32)**—Identifies the attribute to be monitored—usually the value.
- **SamplingInterval (double)**—The fastest rate at which the MonitoredItem should be accessed and evaluated.
- **QueueSize (uint32)**—Requested size of the MonitoredItem queue.
• **DiscardOldest** *(boolean)*—Indicates whether the oldest *Notification* in the queue shall be discarded when the queue is full. If set to *false*, the last added *Notification* shall be replaced.

• **DataChangeFilter** *(DataChangeFilter as defined in Table 8.3)*—Configures the conditions under which a *DataChange Notification* shall be reported.

• **AggregateFilter** *(AggregateFilter as defined in Table 8.3)*—Defines an aggregate function to calculate the values to be returned. Only one filter can be applied at a time.

Note that, depending on the use case, two possible monitoring filters that may be applied to a *DataItem*: *DataChangeFilter* and *AggregateFilter*. A *DataItem* may define one and only one of these filters—they shall not be combined.

Each *EventItem* contains the following configuration parameters:

• **NodeId** *(NodeId as defined in Table 8.2)*—Identifies the *Node* providing the *Event* within the AddressSpace of an OPC UA Server.

• **SamplingInterval** *(double)*—The fastest rate at which the *Event* should be accessed and evaluated.

• **QueueSize** *(uint32)*—Requested size of the *MonitoredItem* queue.

• **DiscardOldest** *(boolean)*—Indicates whether the oldest *Notification* in the queue shall be discarded when the queue is full. If set to false, then the last added *Notification* shall be replaced.

• **EventFilter** *(EventFilter as defined in Table 8.3)*—Provides a way to filter the types of *Events* to be reported, as well as the fields within each *Event* that will be part of the *Notification* message.

Table 8.12 provides the IDL representation for *DataItem* and *EventItem*.

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition (IDL Equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MonitoredItem</td>
<td>enum MonitoredItemKind {&lt;br&gt; DATA_MONITORED_ITEM,&lt;br&gt; EVENT_MONITORED_ITEM }&lt;br&gt;@nested&lt;br&gt;union MonitoredItem switch (MonitoredItemKind) {&lt;br&gt; case DATA_MONITORED_ITEM:&lt;br&gt; DataItem data_item;&lt;br&gt; case EVENT_MONITORED_ITEM:&lt;br&gt; EventItem event_item;&lt;br&gt; }</td>
</tr>
<tr>
<td>DataItem</td>
<td>@nested&lt;br&gt;struct DataItem {&lt;br&gt;    NodeId node_id;&lt;br&gt;    uint32 attribute_id;&lt;br&gt;    double sampling_interval;&lt;br&gt;    uint32 queue_size;&lt;br&gt;    boolean discard_oldest;&lt;br&gt;    // Only one (or none) of the following filter kinds&lt;br&gt;    // can be applied at a time&lt;br&gt;    @optional DataChangeFilter data_change_filter;&lt;br&gt;    @optional AggregateFilter aggregate_filter;&lt;br&gt; }</td>
</tr>
<tr>
<td>EventItem</td>
<td>@nested&lt;br&gt;struct EventItem {</td>
</tr>
</tbody>
</table>
### 8.4.2.3 DDS Outputs

DDS Outputs provide the means to propagate NotificationMessages over DDS. They map a set of Data or Event MonitoredItems from an OPC UA Inputs\(^7\) to a DDS Topic and create the necessary entities to update DDS applications interested in these NotificationMessages.

![Diagram of DDS Output Definition](image)

**Figure 8.4: DDS Output Definition**

Table 8.13 provides an IDL representation for a DDS Output.

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition (IDL Equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DdsOutput</td>
<td>@nested</td>
</tr>
<tr>
<td></td>
<td>struct DdsOutput {</td>
</tr>
<tr>
<td></td>
<td>string name;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

\(^7\) A DDS Output may include MonitoredItems associated with multiple OPC UA Inputs.
8.4.2.3.1 Output Name

Every DDS Output is given a name that identifies it within the mapping section.

8.4.2.3.2 DDS DomainParticipants

The Gateway must refer to a DomainParticipant in order to create the Topics and endpoints capable of propagating OPC UA DataChanges and Events over DDS. A DomainParticipant may be used by different outputs, different OPC UA to DDS Bridges, and different DDS to OPC UA Bridges; therefore, DomainParticipants are annotated as @external to indicate DDS Outputs shall use references to either already existing DomainParticipants or references to newly created objects if they do not exist.

The definition of a DomainParticipant shall only expose a subset of the functionality of DomainParticipants described in the DDS PIM [DDS]; in particular, the following configuration parameters shall be exposed:

- domain_id—Identifies the Domain DDS Outputs associated with the DomainParticipants will bind to.
- register_types—List of types to be registered. These may later be associated with the DDS Topics created in the context of a DDS Output.
- participant_qos—QoS settings of the DomainParticipant to be instantiated by the Gateway.

Table 8.14 provides the IDL definition of a DDS DomainParticipant in the context of the Gateway configuration.

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition (IDL Equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>@external DdsDomainParticipant domain_participant_ref;</td>
<td></td>
</tr>
<tr>
<td>string topic_name;</td>
<td></td>
</tr>
<tr>
<td>string registered_type_name;</td>
<td>＠optional DDS::DataWriterQos datwriter_qos;</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.14: DDS DomainParticipant Definition

8.4.2.3.3 Topic Name

Specifies the name of the Topic that will be used to update the value of the received MonitoredItems.

8.4.2.3.4 Registered Type Name

Specifies the typename of the Topic associated with the OPC UA Output. The type shall have been registered with the DomainParticipant the DDS Output is referencing.
8.4.2.3.5 DataWriterQos

Configures the DDS DataWriter that is instantiated upon the creation of the DDS Output to publish data samples associated with NotificationMessages.

8.4.2.4 Input/Output Mappings

Input/output mappings provide the means to configure many-to-many correspondences between MonitoredItems of OPC UA Inputs (DataItems and EventItems) and DDS Topics of DDS Outputs. In other words, it allows users of the OPC UA/DDS Gateway to route data from OPC UA to DDS.

Figure 8.5: Input/Output Mapping Definition

MonitoredItems associated with an OPC UA Input may be propagated to different DDS Outputs. For DataItems, the Gateway provides the means to map a DataItem (identified by its name) to a specific Topic field in one or more DDS Outputs. In the case of EventItems, the Gateway provides the means to map an element of the EventFieldList (i.e., an
EventField) to a specific Topic field in one or more DDS Outputs. Moreover, input/output mappings provide the means to assign constant values to specific fields of a DDS Topics in one or more DDS Outputs.

Table 8.15 provides the IDL definition of an input/output mapping.

Table 8.15: Input/Output Mapping Definition

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition (IDL Equivalent)</th>
</tr>
</thead>
</table>
| InputOutputMapping        | @nested struct InputOutputMapping {
|                           |     sequence<Assignment> assignments; |
| Assignment                | @nested struct Assignment {
|                           |     @external DdsOutput dds_output_ref;
|                           |     @external OpcUaInput opcua_input_ref;
|                           |     sequence<FieldAssignment> field_assignments; |
| FieldAssignment           | enum AssignmentKind {
|                           |     DATA_ITEM_ASSIGNMENT,
|                           |     EVENT_FIELD_ASSIGNMENT,
|                           |     CONSTANT_VALUE_ASSIGNMENT
|                           | };
|                           |     struct DataItemRef {
|                           |     string data_item_name;
|                           | };
|                           |     struct EventFieldRef {
|                           |     string event_name;
|                           |     uint32 event_field_index;
|                           | };
|                           |     @nested union AssignmentInput switch (AssignmentKind) {
|                           |         case DATA_ITEM_ASSIGNMENT:
|                           |             DataItemRef data_item;
|                           |         case EVENT_FIELD_ASSIGNMENT:
|                           |             EventFieldRef event_field;
|                           |         case CONSTANT_VALUE_ASSIGNMENT:
|                           |             Variant constant_value;
|                           | };
|                           |     @nested struct FieldAssignment {
|                           |         string dds_output_field_ref; // name of output field
|                           |         @optional @external OpcUaInput opcua_input_ref;
|                           |         AssignmentInput assignment_input; |

As shown above, an InputOutputMapping is a sequence of assignments, which apply to a specific DDS Output referenced via dds_output_ref. Each assignment to a DDS Output is also linked to an OPC UA Input via the opcua_input_ref attribute. This implies that all DataItems and EventFields assigned are assumed to belong to MonitoredItems of the given Input.

8 (Non-normative) This mapping model is extremely flexible; however, users of the OPC UA/DDS Gateway should avoid combining MonitorItems of different kinds in the same DDS Output. That is, they should include DataItems or EventItems, but not both.
Every FieldAssignment definition shall provide the fully-qualified name of the member of the Topic type via the dds_output_field_ref attribute. The fully-qualified name shall be represented according to the following syntax: `<member_name> [. <nested_member_name>]`. Optionally, users may provide an OPC UA Input different than the default one specified in the InputOutputMapping declaration. This implicitly enables a DDS Output to publish items from different OPC UA Inputs.

Lastly, AssignmentInput refers to the source of information that shall be assigned. That is, it provides a reference to the DataItem, EventField, or constant that shall the field shall be assigned.

- In the case of DataItems, DataItemRef provides the name of the DataItem from the OPC UA Input that shall be assigned.
- In the case of EventItems, EventItemRef provides the name of the Event and the position in the EventFieldList that shall be assigned.
- In the case of constants, the specific constant to be assigned in the form of a Variant that can take any possible value.

### 8.4.3 OPC UA Subscription Mapping Behavior

This clause describes the OPC UA Subscription Mapping behavior. That is, how the OPC UA/DDS Gateway shall handle NotificationMessages received by the OPC UA Inputs and assign them to DDS Outputs according to the Input/Output mapping rules so that they can be propagated over DDS.

It is important to note that it is up to implementers of this specification to decide when to trigger DDS publications (i.e., when to call write() on the underlying DataWriters) as a response to these input. This specification focuses on the mapping behavior rather than on the necessary optimization strategies.

#### 8.4.3.1 Constant Assignment

In the model specified in sub clause 8.4.2.4, constants are defined as Variants, which—according to the mapping rules specified in clause 8.2.2—makes it impossible to directly assign a Variant to a DDS Output field of any type different than Variant. Therefore, when assigning a constant to a DDS Output field, Variants shall be mapped into the equivalent type following the rules specified in sub clause 8.4.3.3.

The assignment value of a constant value shall be performed only once upon the instantiation of a DDS Output. The DDS Output field shall be compatible with the type deduced from the Variant mapping rules specified in 8.4.3.3 (i.e., shall be safely cast to the type of the DDS Output field); otherwise, the Gateway shall report an error. The mechanism to report errors to the user is out of the scope of this specification.

#### 8.4.3.2 NotificationMessage Assignment

As explained in sub clause 8.4.1.3, NotificationMessages received by OPC UA Clients¹ contain a sequence of NotificationData objects that represent DataChange Notifications, Event Notifications, or StatusChangeNotifications.

This sub clause describes how to assign each Notification to the corresponding DDS Output field.

¹In the case of the Gateway, these are the internal OPC UA Clients that every OPC UA Input uses to create subscriptions and to add MonitoredItems)
8.4.3.2.1 DataChange Notification Assignment

DataChangeNotification messages contain a sequence of MonitoredItemNotification with every monitored DataItem that has changed. The Gateway shall iterate the sequence and process every MonitoredItemNotification as follows:

1. Every MonitoredItemNotification contains an IntegerId value named client_handle, which shall be used to correlate the item to one of the DataItems in the list of MonitoredItems associated with the current OPC UA Input (i.e., the Input associated with the Subscription session and the Client).

2. Once the DataItem has been identified, the Gateway shall lookup the DDS Outputs to be updated with the new value according to the assignments specified in InputOutputMapping.

3. Next, the OPC UA/DDS Gateway shall analyze the DataValue value of the MonitoredItemNotification, which contains a Variant with the real value.

4. Finally, the Gateway shall assign the Variant value—mapped according to the rules specified in 8.4.3.3—to the DDS type field of every DDS Output field (i.e., every DDS type field associated with a DDS Output) identified in 2. If the value cannot be cast, the Gateway shall report an error.

8.4.3.2.2 EventField Assignments

EventNotificationList messages contain a sequence of EventFieldList, where each element represents an Event that has been triggered. The Gateway shall iterate the sequence and process every EventFieldList as follows:

1. Every EventFieldList contains an IntegerId value named client_handle, which shall be used to correlate the Event to one of the Events in the list of MonitoredItems associated with the current OPC UA Input (i.e., the Input associated with the Subscription session and the Client).

2. Once the Event (the EventItem) has been identified, the Gateway shall iterate the sequence of EventFields (event_fields) in the EventFieldList and lookup the DDS Outputs to be updated with the new value according to the assignments specified in InputOutputMapping. In other words, it shall therefore check the combination of event_name and event_field_index that conform an EventFieldRef in every DDS Output.

3. Next, the OPC UA/DDS Gateway shall analyze the value of every EventField (i.e., every element of the event_fields sequence), which is represented as a BaseDataType—a typedef of a Variant.

4. Finally, the Gateway shall assign the BaseDataType value—mapped according to the rules specified in 8.4.3.3 for Variants—to the DDS type field of every DDS Output field (i.e., every DDS type field associated with a DDS Output) identified in 2.

8.4.3.2.3 StatusChangeNotifications

StatusChangeNotifications are used to report changes in the status of a Subscription. The mapping of this type of Notifications is out of the scope of this specification—it is up to the implementers of this specification to decide how to use StatusChangeNotifications.

8.4.3.3 Simplified Mapping of OPC UA Variant Types

To simplify of mapping for OPC UA Variants to equivalent DDS Types that shall be applied when casting the value of DataItems and EventFields; this mapping requires implementations of the OPC UA/DDS Gateway to evaluate the value of array_dimensions of the Variant to determine whether the value is a scalar, an array, or a multi-dimensional array; and the corresponding DDS according to the following rules:
• If the value is a scalar, the value shall be mapped to the equivalent type defined in sub clause 8.2 (e.g., `int32` or its alias `Int32`).

• If the value is a one-dimensional array, then the value shall be mapped to a DDS sequence of the equivalent type for a scalar. This specification defines alias types for each of these sequences (e.g., `Int32Array` as a shortcut for `sequence<int32>`).

• If the value is a multi-dimensional array, then the value is mapped to a structure containing: a one-dimensional DDS sequence of equivalent type for the scalar value, and a sequence of `uint32` to represent the length of every dimension in the multi-dimensional array (e.g., `Int32Matrix`).

Table 8.16 shows the specific mapping for all the different combinations of array dimensions and Variant Values.

<table>
<thead>
<tr>
<th>Array Dimensions</th>
<th>Variant Type</th>
<th>DDS Type (IDL equivalent)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>If <code>array_dimensions</code> is an empty zero-length sequence, the Variant type is mapped to the equivalent type.</td>
<td>Boolean</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SByte</td>
<td>int8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Byte</td>
<td>uint8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Int16</td>
<td>int16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UInt16</td>
<td>uint16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Int32</td>
<td>int32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UInt32</td>
<td>uint32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Int64</td>
<td>int64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UInt64</td>
<td>uint64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Float</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td></td>
<td>String</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DateTime</td>
<td>DateTime as defined in OPC UA Type System Mapping (Table 8.2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Guid</td>
<td>Guid as defined in OPC UA Type System Mapping (Table 8.2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ByteString</td>
<td>ByteString as defined in OPC UA Type System Mapping (Table 8.2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XElement</td>
<td>XElement as defined in OPC UA Type System Mapping (Table 8.2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NodeId</td>
<td>NodeId as defined in OPC UA Type System Mapping (Table 8.2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExpandedNodeId</td>
<td>ExpandedNodeId as defined in OPC UA Type System Mapping (Table 8.2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>StatusCode</td>
<td>StatusCode as defined in OPC UA Type System Mapping (Table 8.2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>QualifiedName</td>
<td>QualifiedName as defined in OPC UA Type System Mapping</td>
<td></td>
</tr>
</tbody>
</table>

10 All these types appear inside the IDL module `OMG::DDSOPCUA::OPCUA2DDS`. 

---

### Table 8.16: Simplified Mapping of OPC UA Variant Type to DDS Types

- **Array Dimensions**
  - If the value is a scalar, the value shall be mapped to the equivalent type defined in sub clause 8.2 (e.g., `int32` or its alias `Int32`).
  - If the value is a one-dimensional array, then the value shall be mapped to a DDS sequence of the equivalent type for a scalar. This specification defines alias types for each of these sequences (e.g., `Int32Array` as a shortcut for `sequence<int32>`).
  - If the value is a multi-dimensional array, then the value is mapped to a structure containing: a one-dimensional DDS sequence of equivalent type for the scalar value, and a sequence of `uint32` to represent the length of every dimension in the multi-dimensional array (e.g., `Int32Matrix`).

- **Table 8.16 shows the specific mapping for all the different combinations of array dimensions and Variant Values.**
<table>
<thead>
<tr>
<th>Array Dimensions</th>
<th>Variant Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><em>(Table 8.2).</em></td>
</tr>
<tr>
<td>LocalizedText</td>
<td>LocalizedText</td>
<td>as defined in OPC UA Type System Mapping <em>(Table 8.2).</em></td>
</tr>
<tr>
<td>ExtensionObject</td>
<td>ExtensionObject</td>
<td>as defined in OPC UA Type System Mapping <em>(Table 8.2).</em></td>
</tr>
</tbody>
</table>

**Table:**

If `array_dimensions` is a sequence of length one, Variant Types are mapped to an array of the equivalent type.

<table>
<thead>
<tr>
<th>Variant Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BooleanArray</td>
<td><code>sequence&lt;boolean&gt;</code></td>
</tr>
<tr>
<td>SByteArray</td>
<td><code>sequence&lt;int8&gt;</code></td>
</tr>
<tr>
<td>ByteArray</td>
<td><code>sequence&lt;uint8&gt;</code></td>
</tr>
<tr>
<td>Int16Array</td>
<td><code>sequence&lt;int16&gt;</code></td>
</tr>
<tr>
<td>UInt16Array</td>
<td><code>sequence&lt;uint16&gt;</code></td>
</tr>
<tr>
<td>Int32Array</td>
<td><code>sequence&lt;int32&gt;</code></td>
</tr>
<tr>
<td>UInt32Array</td>
<td><code>sequence&lt;uint32&gt;</code></td>
</tr>
<tr>
<td>Int64Array</td>
<td><code>sequence&lt;int64&gt;</code></td>
</tr>
<tr>
<td>UInt64Array</td>
<td><code>sequence&lt;uint64&gt;</code></td>
</tr>
<tr>
<td>FloatArray</td>
<td><code>sequence&lt;float&gt;</code></td>
</tr>
<tr>
<td>DoubleArray</td>
<td><code>sequence&lt;double&gt;</code></td>
</tr>
<tr>
<td>StringArray</td>
<td><code>sequence&lt;string&gt;</code></td>
</tr>
<tr>
<td>DateTimeArray</td>
<td><code>sequence&lt;DateTime&gt;</code></td>
</tr>
<tr>
<td>GuidArray</td>
<td><code>sequence&lt;Guid&gt;</code></td>
</tr>
<tr>
<td>ByteStringArray</td>
<td><code>sequence&lt;ByteString&gt;</code></td>
</tr>
<tr>
<td>XmlElementArray</td>
<td><code>sequence&lt;XmlElement&gt;</code></td>
</tr>
<tr>
<td>NodeIdArray</td>
<td><code>sequence&lt;NodeId&gt;</code></td>
</tr>
<tr>
<td>ExpandedNodeIdArray</td>
<td><code>sequence&lt;ExpandedNodeId&gt;</code></td>
</tr>
<tr>
<td>StatusCodeArray</td>
<td><code>sequence&lt;StatusCode&gt;</code></td>
</tr>
<tr>
<td>QualifiedNameArray</td>
<td><code>sequence&lt;QualifiedName&gt;</code></td>
</tr>
<tr>
<td>LocalizedTextArray</td>
<td><code>sequence&lt;LocalizedText&gt;</code></td>
</tr>
<tr>
<td>ExtensionObjectArray</td>
<td><code>sequence&lt;ExtensionObject&gt;</code></td>
</tr>
</tbody>
</table>

If `array_dimensions` is a sequence of length greater than one, Variant types are mapped to a structure that contains: (1) an array of the equivalent type, and `array_dimensions`.

```cpp
// BooleanMatrix
struct BooleanMatrix {
    BooleanArray array;
    sequence<uint32> array_dimensions;
};
```

```cpp
// SByteMatrix
struct SByteMatrix {
    SByteArray array;
    sequence<uint32> array_dimensions;
};
```

```cpp
// ByteMatrix
struct ByteMatrix {
    ByteArray array;
    sequence<uint32> array_dimensions;
};
```
<table>
<thead>
<tr>
<th>Array Dimensions</th>
<th>Variant Type</th>
<th>DDS Type (IDL equivalent)</th>
</tr>
</thead>
</table>
| Int16Matrix      | struct Int16Matrix{  
|                  |   Int16Array array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| UInt16Matrix     | struct UInt16Matrix{  
|                  |   UInt16Array array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| Int32Matrix      | struct Int32Matrix{  
|                  |   Int32Array array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| UInt32Matrix     | struct UInt32Matrix{  
|                  |   UInt32Array array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| Int64Matrix      | struct Int64Matrix{  
|                  |   Int64Array array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| UInt64Matrix     | struct UInt64Matrix{  
|                  |   UInt64Array array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| FloatMatrix      | struct FloatMatrix{  
|                  |   FloatArray array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| DoubleMatrix     | struct DoubleMatrix{  
|                  |   DoubleArray array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| StringMatrix     | struct StringMatrix{  
|                  |   StringArray array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| DateTimeMatrix   | struct DateTimeMatrix{  
|                  |   DateTimeArray array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| GuidMatrix       | struct GuidMatrix{  
|                  |   GuidArray array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| ByteStringMatrix | struct ByteStringMatrix{  
|                  |   ByteStringArray array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| XmlElementMatrix | struct XmlElementMatrix{  
|                  |   XmlElementArray array;  
|                  |   sequence<uint32> array_dimensions;  
|                  | } | |
| NodeIdMatrix     | struct NodeIdMatrix{  
|                  | } | |
### 8.4.4 Implementation Considerations

#### 8.4.4.1 OPC UA Implementation Considerations

The mapping of OPC UA Subscriptions specified in this chapter requires the OPC UA/DDS Gateway to embed one or more OPC UA Clients. These OPC UA Clients shall be capable of:

- Connecting to OPC UA Servers using the Discovery, SecureChannel, and Session Service Sets.
- Creating Subscriptions and issuing Publish and Republish requests using the Subscription Service Set.
- Creating MonitoredItems using the MonitoredItem Service Set.

To comply with all the requirements listed above, implementers of this specification shall use OPC UA Clients compliant with the Standard UA Client Profile defined in sub clause 6.5.121 of [OPCUA-07]. Alternatively, implementers of this specification may use an OPC UA Client that is not fully compliant with the Standard UA Client Profile, but complies with the following Client Facets specified in [OPCUA-07]:

- Core Client Facet
- Base Client Behavior Facet
- Discovery Client Facet
- DataChange Subscriber Client Facet

Additionally, OPC UA Clients (whether they are compliant with Standard UA Client Profile or compliant with the required Client Facets listed above) shall support an extra facet to configure Event subscriptions: the Event Subscriber Client Facet defined in sub clause 6.5.76 of [OPCUA-07].

Consequently, compliant implementations of this specification shall be built upon an OPC UA implementation capable of passing the conformance tests specified for those profiles and facets by the OPC Foundation.
Lastly, it is important to note that implementers of this specification may need to configure the underlying OPC UA Clients to satisfy the requirements of the remote OPC UA Servers in terms of authentication, access control, and encryption using the mechanisms provided by the OPC UA Security Model [OPCUA-02]. Depending on the requirements of the remote OPC UA Servers, OPC UA Clients may need to support additional security-related facets from [OPCUA-07].

8.4.4.2 DDS Implementation Considerations

To implement the mappings specified in this chapter OPC UA/DDS Gateway shall use a DDS implementation complaint with:
- Minimum Profile of [DDS].
- Statements listed in clause 8.4.2 of [DDSI-RTPS].

As specified in the rest of clauses dealing with DDS and OPC UA integration, the Gateway shall be capable of dealing with two different security models: the OPC UA Security Model on one end and the DDS Security Model on the other end. Each security model shall be configured separately depending on the needs of the end user of the OPC UA/DDS Gateway.
9  DDS to OPC UA Bridge

This chapter defines the DDS to OPC UA Bridge, which enables OPC UA Clients to browse, read, write, and receive notifications on status changes in the DDS Global Data Space. In other words, it enables OPC UA Clients to participate as first-class citizens in the DDS Global Data Space.

9.1  Overview (non-normative)

Figure 9.1 shows an example of the OPC UA/DDS Gateway implementing the DDS to OPC UA Bridge.

Figure 9.1: DDS to OPC UA Bridge Overview

On one side of the Gateway, a set of DDS DomainParticipants and Endpoints (i.e., DataWriters and DataReaders) construct a view of the DDS Global Data Space by joining to DDS Domains, subscribing to DDS Topics, and receiving updates on DDS Topic Instances. On the other side of the Gateway, an OPC UA Server represents in its AddressSpace that view of DDS Global Data Space using Nodes and References as specified in this chapter.

The resulting deployment enables OPC UA Clients to browse the Topics available on a certain Domain using the View Service Set, subscribe to data updates on specific instances of those Topics using the Subscription and MonitoredItems Service Sets, and read or write updates to those instances using the Attribute Service Set.

The chapter is organized as follows:

- Sub clause 9.2 defines a mapping of the DDS type system to OPC UA.
- Sub clause 9.3 defines an OPC UA Information Model to represent the DDS Global Data Space using OPC UA Nodes and References.

9.2  DDS Type System Mapping

This clause defines a complete mapping of the DDS Type System to OPC UA.
9.2.1  Primitive Types

9.2.1.1  Overview (non-normative)

DDS provides a rich set of primitive types that cover the basic data types used in most common programming languages. These include boolean types, byte types, integral types of various lengths, floating point types of various precisions, and single-byte and wide-character types.

OPC UA provides also a rich set of primitive types equivalent, in most cases, to those that are part of the DDS Type System. The only exception is the absence of a 128-bit floating point type, which can nevertheless be represented using other built-in types.

Because there is a one-to-one correspondence between primitive types in DDS and OPC UA, it is unnecessary to define new OPC UA DataTypes, ObjectTypes, or VariableTypes represent DDS primitive types\(^{11}\). Therefore, this clause focuses on specifying how to create Variables of equivalent types.

9.2.1.2  Mapping

**DDSOPCU-3: Improve readability of a number of figures**

Primitive types shall be represented as Nodes of Variable NodeClass in the AddressSpace an OPC UA Server as shown in Figure 9.2, Figures 9.2, 9.3, and 9.4. These Variable Nodes may become components of complex VariableTypes or ObjectTypes as a result of the mappings specified in this document.

---

\(^{11}\) As defined below, there are workarounds to define the unsupported Float128 type.
Table 9.1 specifies the Attributes every Variable Node shall be instantiated with.

### Table 9.1: Primitive Type Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td>BrowseName shall be a string matching the DDS variable with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>&lt;ValueRank&gt;</td>
<td>ValueRank shall be set as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the Variable represents a Primitive Type, ValueRank shall be set to 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the Variable represents an Array of Primitive Types, ValueRank shall be set to the number of dimensions of the array (see sub clause 9.2.5.1).</td>
</tr>
</tbody>
</table>

---

**Figure 9.3:** Primitive Types Mapping to OPC UA—Floating Point Types

**Figure 9.4:** Primitive Types Mapping to OPC UA—Boolean, Byte, and Char Types
If the `Variable` represents a Sequence of Primitive Types, `ValueRank` shall be set to 1 (see sub clause 9.2.5.2).

ArrayDimensions: `[...] | <NULL>`

`ArrayDimensions` shall be set as follows:

- If the `Variable` represents a Primitive Type, `ArrayDimensions` shall be set to `NULL`.
- If the `Variable` represents an Array of Primitive Types, `ArrayDimensions` shall be set as specified in sub clause 9.2.5.1.
- If the `Variable` represents a Sequence of Primitive Types, `ArrayDimensions` shall be set as specified in sub clause 9.2.5.2.

DataType: `<NodeId>`

`DataType` shall be set to the `NodeId` of the equivalent OPC UA primitive data type. The mapping between DDS Primitive Types and OPC UA Primitive Types is specified in Table 9.2.

For example, if the DDS primitive type is a `Boolean`, `DataType` shall be the `NodeId` of the OPC UA built-in type `Boolean`.

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDefinition</td>
<td>VariableType</td>
<td>BaseDataVariable Type</td>
<td>Because this is a simple <code>DataVariable</code> with no more concrete type definition needs, it shall be defined as a <code>BaseDataVariableType Variable</code>.</td>
</tr>
</tbody>
</table>

Table 9.2 specifies the equivalent OPC UA built-in types for every DDS primitive type.

<table>
<thead>
<tr>
<th>DDS Primitive Type</th>
<th>IDL Equivalent Type</th>
<th>OPC UA Built-in Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>octet</td>
<td>Byte</td>
</tr>
<tr>
<td>Boolean</td>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>Int8&lt;sup&gt;12&lt;/sup&gt;</td>
<td>int8</td>
<td>SByte</td>
</tr>
<tr>
<td>UInt8</td>
<td>uint8</td>
<td>Byte</td>
</tr>
<tr>
<td>Int16</td>
<td>int16</td>
<td>Int16</td>
</tr>
<tr>
<td>UInt16</td>
<td>uint16</td>
<td>UInt16</td>
</tr>
<tr>
<td>Int32</td>
<td>int32</td>
<td>Int32</td>
</tr>
<tr>
<td>UInt32</td>
<td>uint32</td>
<td>UInt32</td>
</tr>
<tr>
<td>Int64</td>
<td>int64</td>
<td>Int64</td>
</tr>
<tr>
<td>UInt64</td>
<td>uint64</td>
<td>UInt64</td>
</tr>
<tr>
<td>Char8</td>
<td>char</td>
<td>Byte</td>
</tr>
</tbody>
</table>

<sup>12</sup> Int8 and UInt8 have recently been added to [IDL]. Even though they are not part of the current DDS Type System specified in [DDS-XTYPES], they are planned for the next revision of the specification, and they are therefore added to the table for completeness.
### 9.2.1.3 Example (non-normative)

Let us use the following example to illustrate the mapping of a simple 32-bit integer value to an OPC UA Variable.

A 32-bit integer variable `x`, member of a structure type, is represented in IDL as follows:

```idl
struct StructuredType {
    int32 my_integer;
};
```

To represent `my_integer` in OPC UA, we shall create a `Variable` following the rules specified in Table 9.1.

Figure 9.5 shows the OPC UA `Nodes` and `References` involved in the mapping.

![Diagram](image)

**Figure 9.5: Example of Primitive Type Mapping to OPC UA**

Table 9.3 shows the definition of the `Variable` representing `my_integer`.

#### Table 9.3: Example of Int32 Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>my_integer</code></td>
<td><code>BrowseName</code> matches the name of the original DDS variable: <code>my_integer</code>.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td><code>ValueRank</code> of -1 to indicate the <code>Variable</code> contains a scalar <code>Value</code>.</td>
</tr>
<tr>
<td>DataType</td>
<td><code>Int32</code></td>
<td><code>NodeId</code> of <code>Int32</code>, which is the type equivalent to a DDS 32-bit integer.</td>
</tr>
<tr>
<td>Value</td>
<td><code>&lt;Int32&gt;</code></td>
<td>A valid 32-bit integer value (e.g., 13). If the <code>Variable</code> is used in the definition of a complex <code>VariableType</code> or <code>ObjectType</code>, <code>Value</code> may be overwritten by the instance of the corresponding type.</td>
</tr>
</tbody>
</table>

To store the `Float128` value, the length of the equivalent `ByteString` shall be 16.
9.2.2 String Types

9.2.2.1 Overview (non-normative)

String Types are ordered one-dimensional variable-sized collections of characters [DDS-XTYPES]. The DDS Type System includes two character types: Char8 and Char16. Therefore, it specifies two equivalent string types composed of these character types: String8 and String16.

In CDR, String8 strings—commonly referred to as strings—are represented using UTF-8 character encoding, where characters take from one to four bytes of space. In contrast, String16 strings—commonly referred to as wstrings or wide strings—are represented using UTF-16 character encoding, where characters take two bytes if they are part of the Basic Multilingual Plane (BMP) and four bytes otherwise. [DDS-XTYPES] limits the characters that may be used in a String16 string to those in the BMP. As a result, every Unicode character in a String16 always takes two bytes.

OPC UA specifies two built-in String types: String and ByteString [OPCUA-03]. Strings are used to represent UTF-8 encoded strings. Therefore, DDS string types can be directly mapped to OPC UA Strings. In contrast, ByteStrings represent opaque sequence of bytes. Because OPC UA does not provide an explicit way of representing UTF-16-encoded strings, wide strings shall be mapped to ByteStrings where every character is represented as a two-byte pair.

9.2.2.2 Mapping

String types shall be represented as Nodes of Variable NodeClass in the AddressSpace of an OPC UA Server as shown in Figure 9.6. These Variable Nodes may become components of complex VariableTypes or ObjectTypes as a result of the mappings specified in this document.

[Diagram of String Types Mapping to OPC UA]

Figure 9.6: String Types Mapping to OPC UA
Table 9.4 defines the mapping of the String8 type to an OPC UA Variable.

**Table 9.4: String8 (String) Variable Definition**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td>BrowseName shall be a string matching the name of the DDS variable with the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td>Because variables of String8 (strings) represent scalar values, they shall</td>
</tr>
<tr>
<td></td>
<td></td>
<td>have a ValueRank of -1.</td>
</tr>
<tr>
<td>DataType</td>
<td>String</td>
<td>NodeId of the OPC UA built-in type equivalent to String8: String.</td>
</tr>
</tbody>
</table>

References

<table>
<thead>
<tr>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VariableType</td>
<td>BaseDataVariableType</td>
<td>Because this is a simple DataVariable with no more concrete type definition needs, it shall be defined as a BaseDataVariableType Variable.</td>
</tr>
</tbody>
</table>

Table 9.5 defines a mapping of the String16 type to an OPC UA Variable.

**Table 9.5: String16 (Wide String) Variable Definition**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td>BrowseName shall be a string matching the name of the DDS variable with the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td>Because variables of String16 type (wide strings) represent scalar values,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>they shall have a ValueRank of -1.</td>
</tr>
<tr>
<td>DataType</td>
<td>ByteString</td>
<td>NodeId of the OPC UA built-in type equivalent to String16: ByteString. In</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the equivalent ByteString, each Unicode character is represented as two</td>
</tr>
<tr>
<td></td>
<td></td>
<td>consecutive bytes; therefore, the length of the ByteString shall be the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number of characters in the wide string times two.</td>
</tr>
</tbody>
</table>

References

<table>
<thead>
<tr>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VariableType</td>
<td>BaseDataVariableType</td>
<td>Because this is a simple DataVariable with no more concrete type definition needs, it shall be defined as a BaseDataVariableType Variable.</td>
</tr>
</tbody>
</table>

9.2.2.3 Example (non-normative)

Let us use the following example to illustrate the mapping of a string type to an OPC UA Variable.

A String8 variable `my_string`, member of a structure type, is represented in IDL as follows:

```cpp
struct StructuredType {
    string my_string;
};
```

To represent `my_string` in OPC UA, we shall create a Variable following the rules specified in Table 9.4.

---

14 Indeed, they are a special kind of scalar values that contain a collection of characters.
Figure 9.7 shows the OPC UA Nodes and References involved in the mapping.

Table 9.6 shows the definition of the Variable representing my_string.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>my_string</td>
<td>browseName matches the name of the original DDS Variable: my_string.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td>ValueRank of -1 to indicate the Variable contains a scalar Variable.</td>
</tr>
<tr>
<td>DataType</td>
<td>String</td>
<td>NodeId of String, the equivalent type for a DDS String8.</td>
</tr>
<tr>
<td>Value</td>
<td>&lt;String&gt;</td>
<td>A valid string value (e.g., “Julia”). When the Variable is used in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>definition of a complex VariableType or ObjectType, Value may be overwritten</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by the instance of the corresponding Instance Type.</td>
</tr>
<tr>
<td>References</td>
<td>NodeClass</td>
<td>BrowseName</td>
</tr>
<tr>
<td>HasTypeDefinition</td>
<td>VariableType</td>
<td>BaseDataVariable Type Because this is a simple DataVariable with no more</td>
</tr>
<tr>
<td></td>
<td></td>
<td>human-readable representation of the enumeration when the enumeration is zero-based and has no gaps.</td>
</tr>
</tbody>
</table>
• The EnumValues Property defines an array of EnumValueType, which is a Structure DataType that holds: (1) an integer representation of the enumerated value (Int64 in this case); (2) a display name for the human-readable representation of the enumerated value (LocalizedText); and (3) a localized description of the enumerated value (LocalizedText—may be set to an empty string when no description is available).

9.2.3.1.2 Mapping

Every DDS Enumeration type definition shall be mapped to an OPC UA Enumeration DataType. Instances of DDS Enumeration Types, such as members of Aggregated Types and elements of Collection Types, shall be mapped to Variables of the corresponding OPC UA Enumeration DataType as show in Figure 9.8.

![Figure 9.8: Enumeration Types Mapping to OPC UA](image)

The OPC UA Enumeration DataType shall be defined as a subtype of the standard Enumeration DataType as specified in Table 9.7.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;EnumerationTypeName&gt;DataType</td>
<td>Enumeration</td>
<td>The equivalent Enumeration DataType shall be a subtype of the standard Enumeration DataType.</td>
</tr>
</tbody>
</table>

The DataType shall be named according to the following convention: <EnumerationTypeName>DataType. Where <EnumerationTypeName> corresponds to the name of the original DDS Enumeration Type. For example, if the name of the original Enumeration Type is TemperatureKind, then the OPC UA DataType shall be named TemperatureKindDataType.

Because DDS Enumeration Types may be not zero-based and may have gaps, <EnumerationTypeName>DataType shall include a reference to an EnumValues Property. This property shall be defined as an array of EnumValueType, where every element shall represent an enumerated literal as follows:

• Value shall be set to the enumerated literal value.
• DisplayName shall be set to the string representation of the enumerated literal constant.
• Description may be set to any specification-specific string.
Variables of `<EnumerationTypeName>DataType` shall be defined as specified in Table 9.8.

### Table 9.8: Enumeration Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td><code>BrowseName</code> shall be a string matching the name of the DDS variable with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td><code>ValueRank</code> shall be -1, indicating that <code>Value</code> is a scalar.</td>
</tr>
<tr>
<td>Value</td>
<td>&lt;Int32&gt;</td>
<td>Integer value of the Enumeration (e.g., 2).</td>
</tr>
<tr>
<td>DataType</td>
<td>&lt;NodeId&gt;</td>
<td><code>DataType</code> shall point to the <code>NodeId</code> of <code>&lt;EnumerationTypeName&gt;DataType</code>.</td>
</tr>
</tbody>
</table>

9.2.3.1.3 Example (non-normative)

Let us use the following example to illustrate the mapping of a common `WorkDays` enumeration type that assigns an integer value to every work day of the week.

`WorkDays` is represented in IDL as follows:

```idl
enum WorkDays {
    @value(1) MONDAY,
    @value(2) TUESDAY,
    @value(3) WEDNESDAY,
    @value(4) THURSDAY,
    @value(5) FRIDAY
}
```

To represent `WorkDays` in OPC UA, we shall define an equivalent `DataType` named `WorkDaysDataType`. Instances of `WorkDaysDataType`, such as a variable of `WorkDays`, shall be represented as Variables in the `AddressSpace` of the OPC UA Server.

Figure 9.9 shows the OPC UA `Nodes` and `References` involved in the mapping.
Table 9.9 shows the equivalent `WorkDaysDataType`.

**Table 9.9: Example of Enumeration DataType Definition**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkDaysDataType</td>
<td>Enumeration</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>BrowseName</th>
<th>Value</th>
</tr>
</thead>
</table>
| HasProperty | EnumValueType[] | EnumValues | [0]  
  
  Value = 1  
  DisplayName = “MONDAY”  
  Description = “I don’t like Mondays!” |
|           |                           |            | [1]  
  
  Value = 2  
  DisplayName = “TUESDAY”  
  Description = “Today is Tuesday!” |
|           |                           |            | [2]  
  
  Value = 3  
  DisplayName = “WEDNESDAY”  
  Description = “Today is Wednesday!” |
|           |                           |            | [3]  
  
  Value = 4  
  DisplayName = “THURSDAY” |
To represent a specific instance of a WorkDays enumeration, we shall create Variables of WorkDaysDataType type as specified in Table 9.8. Table 9.10 shows a variable representing “Monday.”

Table 9.10: Example of Enumeration Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>my_workday</td>
<td>Variable name.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td>The value is a scalar.</td>
</tr>
<tr>
<td>Value</td>
<td>1</td>
<td>Integer value representing “Monday.”</td>
</tr>
<tr>
<td>DataType</td>
<td>WorkDaysDataType</td>
<td>NodeId of the WorkDaysDataType.</td>
</tr>
</tbody>
</table>

References

<table>
<thead>
<tr>
<th>HasTypeDefinition</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VariableType</td>
<td></td>
<td>BaseDataVariable Type</td>
<td>Because this is a simple DataVariable with no more concrete type definition needs, it shall be defined as a BaseDataVariableType Variable.</td>
</tr>
</tbody>
</table>

9.2.3.2 Bitmask Types

9.2.3.2.1 Overview (non-normative)

In DDS, a Bitmask type represents a collection of boolean flags that can be inspected and set individually [DDS-XTYPES]. Bitmasks provide an efficient representation, where every boolean flag is represented with a single bit, rather than with a native boolean value, an integer, or an octet.

Every Bitmask reserves a number of bits (boolean flags) that indicate its bound. The bound of a DDS Bitmask shall be greater than zero and no greater than 64. Each bit is identified by a name and by an index, which is numbered from 0 to bound-1.

In OPC UA bit masks are represented as subtypes of the abstract OptionSet DataType. Every OptionSet is defined as a structure containing two ByteStrings to represent the value and the valid bits [OPCUA-03]:

- Value is an array of bytes representing the bits in the Bitmask. The length depends on the number of bits.
- ValidBits is an array of bytes with the same size as value that represents the bits in the Bitmask that been set. In other words, the bits that have a meaning.

To provide a human-readable representation for every bit in the Bitmask, subtypes of the OPC UA OptionSet DataType shall have an OptionSetValues Property. This property is equivalent to the EnumStrings Property for Enumeration Types (described in sub clause 9.2.3.1.1). It is defined as array of LocalizedText containing the human-readable representation for every bit.
9.2.3.2.2 Mapping

Every DDS Bitmask Type definition shall be mapped to an OPC UA OptionSet DataType. Instances of DDS Bitmask Types, such as members of Aggregated Types and elements of Collection Types, shall be mapped to Variables of the corresponding OPC UA OptionSet DataType as show in Figure 9.10.

The OPC UA OptionSet DataType shall be defined as a subtype of the standard abstract OptionSet DataType as specified in Table 9.11.

Table 9.11: Bitmask DataType Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;BitmaskTypeName&gt;DataType</td>
<td>OptionSet</td>
<td>The equivalent OptionSet DataType shall be a subtype of the standard abstract OptionSet DataType.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The OptionSet shall be named according to the following convention: &lt;BitmaskTypeName&gt;DataType. Where &lt;BitmaskTypeName&gt; corresponds to the name of the original DDS Bitmask Type. For example, if the name of the original Bitmask Type is StatusMask, then the OPC UA DataType shall be named StatusMaskDataType.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;BitmaskTypeName&gt;DataType shall have an OptionSetValues Property. This property shall be represented as an array of LocalizedText of size equal to the Bitmask bound, where every element of the array shall include the string representation of the Bitflag in the position of the corresponding position (whether it has been explicitly set or not).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If no Bitflag has been defined to cover the corresponding position (i.e., if no Bitflag has position x), then the corresponding element of the array shall include the string “UndefinedPosition &lt;PositionNumber&gt;” where &lt;PositionNumber&gt; is the representation in decimal of the position for which no Bitflag has been defined.</td>
</tr>
</tbody>
</table>

References | NodeClass | BrowseName | DataType | TypeDefinition | Modeling Rule |
|------------|-----------|------------|----------|----------------|---------------|

Figure 9.10: Bitmask Types Mapping to OPC UA
Variables of `<BitmaskTypeName>DataType` shall be defined as specified in Table 9.12.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>&lt;String&gt;</code></td>
<td><code>BrowseName</code> shall be a string matching the name of the DDS variable with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td><code>ValueRank</code> shall be -1, indicating that Value is a scalar.</td>
</tr>
</tbody>
</table>
| Value              | `<<BitmaskTypeName>DataType>`               | The Value of the two members of the structure representing `<BitmaskTypeName>DataType` shall be set as follows:  
|                    |                                            | • The `value` `ByteString` shall have a length equal to the bound of the original Bitmask Type.  
|                    |                                            | • The `validBits` `ByteString` shall have a length equal to the bound of the original Bitmask type. |
| DataType           | `<NodeId>`                                 | `DataType` shall point to the `NodeId` of `<BitmaskTypeName>DataType`.         |

9.2.3.2.3 Example (non-normative)

Let us use the following example to illustrate the mapping of a Bitmask type to an OPC UA `Variable`.

A Bitmask with the access permissions in a Unix system is represented in IDL as follows:

```java
@bit_bound(3)
bitmask AccessPermission {
    READ_PERMISSION,
    WRITE_PERMISSION,
    EXECUTE_PERMISSION
};
```

To represent `AccessPermission` in OPC UA, we shall define an equivalent `DataType` named `AccessPermissionDataType`. Instances of `AccessPermissionDataType`, such as `user_permission`, shall be represented as `Variables`. 
Figure 9.11 shows the OPC UA Nodes and References involved in the mapping.

Table 9.13 shows the equivalent AccessPermissionDataType.

Table 9.13: Example of Bitmask DataType Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessPermissionDataType</td>
<td>OptionSet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>BrowseName</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasProperty</td>
<td>LocalizedText[]</td>
<td>OptionSetValues</td>
<td>[0] “READ_PERMISSION”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[1] “WRITE_PERMISSION”</td>
</tr>
</tbody>
</table>

As specified above, for a Bitmask with gaps (i.e., a Bitmask that does not associate a bitflag for a specific position), the array of LocalizedText array shall include predefined strings to indicate that the position is undefined.

For example, for the following Bitmask:

```csharp
@bit_bound(5)
bitmask BitmapWithGaps {
    INITIAL_FIELD,
    @position(2) MIDDLE_FIELD,
    @position(4) LAST_FIELD
};
```

`OptionSetValues` would be set as follows:

- [0] “INITIAL_FIELD”
- [1] “UndefinedPosition_1”
- [2] “MIDDLE_FIELD”
Table 9.14 defines `user_permission`—a *Variable* representing the permission for a specific user.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>user_permission</code></td>
<td>Access permission for a specific user.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td>The value is a scalar.</td>
</tr>
<tr>
<td>Value</td>
<td><code>values:</code></td>
<td><code>values</code> is a <em>ByteString</em> where every element represents the boolean value of a bitflag. In the example, <code>values</code> represents a Bitmask with value 100, indicating that the user has read-only permission.</td>
</tr>
<tr>
<td></td>
<td>[0] “true”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] “false”</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>validValues:</code></td>
<td><code>validValues</code> is a <em>ByteString</em> where every element represents a boolean value indicating whether the position in the Bitmask has been defined. In this case, because all positions have been defined, all elements in <code>values</code> are set to “true”.</td>
</tr>
<tr>
<td></td>
<td>[0] “true”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1] “true”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[2] “true”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[4] “true”</td>
<td></td>
</tr>
<tr>
<td>DataType</td>
<td><code>AccessPermission</code></td>
<td><em>NodeId</em> of <code>AccessPermissionDataType</code>.</td>
</tr>
<tr>
<td>References</td>
<td><code>NodeClass</code></td>
<td><code>BrowseName</code></td>
</tr>
<tr>
<td></td>
<td><code>BaseDataType</code></td>
<td><code>Description</code></td>
</tr>
<tr>
<td>HasTypeDefinition</td>
<td><code>VariableType</code></td>
<td><code>BaseDataType</code> Description Because this is a simple <code>DataVariable</code> with no more concrete definition needs, it shall be defined as a <code>BaseDataType</code> <code>Variable</code>.</td>
</tr>
</tbody>
</table>
9.2.4 Aggregated Types

9.2.4.1 Structure Types

9.2.4.1.1 Overview (non-normative)

In DDS, Structure types are complex types composed of members of any Primitive, String, Collection, Enumerated, or Aggregated type—including other Structure types [DDS-XTYPES].

In OPC UA, Structure types may be represented in different ways. [OPCUA-03] discusses in clause A.4.3 three different approaches for representing structured types:

1. Representing simple members of the Structure type as Variables of simple DataTypes grouped in Objects.
2. Creating Structure DataTypes derived from the standard abstract Structure DataType and instantiating these into a single Variable.
3. Creating both a Structure DataType and a complex VariableType of that DataType including also sub-Variables to represent simple members of the structure.

The first approach provides easy access for generic OPC UA Clients, because every member of the structure is visible in the AddressSpace of the OPC UA Server. However, this approach does not provide a transactional context where the Server can pass directly the structure to the specific OPC UA Client.

The second approach provides such transactional context, but the information exposed by the OPC UA Server cannot be interpreted by generic OPC UA Clients. Furthermore, OPC UA Clients may not access individual items and need to read the whole structure to process a single data item.

The third approach combines the first two approaches: it provides a transactional context and it exposes individual items as Variables that can be separately read by generic OPC UA Clients.

The first structure is more adequate for scenarios in which a transactional context is unnecessary and data items can be interpreted separately, because it simplifies the OPC UA Server logic. (The OPC UA Server needs not offer information in both its native format—structure—and in interpreted format—separate items.) However, in DDS data structures are usually modeled as a whole (e.g., a DataReader must receive the value of longitude and latitude to fully process an instance of a Position Topic composed of both members). The only scenario in which members of a structure could be sent and processed separately would be in that of a structure containing only optional members.

As a result, this specification has chosen to model Structure types following the third approach; that is, providing a Structure DataType, and a VariableType of that DataType including references to sub-variables with simple members of the structure. This approach guarantees that the exposed information can be processed by both generic and specific OPC UA Clients depending on the use case.

9.2.4.1.2 Mapping

Every DDS Structure Type shall be mapped to both an OPC UA Structure DataType and a complex VariableType. Instances of the DDS Structure type shall be represented as Variables of the specified VariableType as shown in Figure 9.12.

---

15 The OPC Unified Architecture Book discusses these options in more detail in Section 3.3.3 “Providing Complex Data Structures.” Further information may be found in a whitepaper entitled OPC UA Information Model Deployment Whitepaper (pp. 17-18), and in the Unified Automation .NET Based OPC UA Client/Server SDK User's Manual.
The Structure DataType shall be defined as a subtype of the standard Structure DataType. It shall be named after the original DDS Structure type according to the following naming convention: `<StructureTypeName>DataType`. Every member of the structure shall be added as a child field where:

- The **field name** shall match the DDS member name, including capitalization.
- The **field type** shall be the member’s OPC UA equivalent type as specified by the mapping rules defined in this chapter.

Table 9.15: Structure DataType Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;StructureTypeName&gt;DataType</code></td>
<td>Structure</td>
<td>Structure representing the DDS structure type.</td>
</tr>
<tr>
<td><code>&lt;MemberName&gt;</code></td>
<td><code>&lt;EquivalentType&gt;</code></td>
<td>First member of the structure. The field name shall be the name of the original DDS structure member. The type shall be the equivalent OPC UA type for the original member of the structure.</td>
</tr>
</tbody>
</table>

...
The VariableType shall be defined as a subtype of BaseDataVariableType and shall be named after the original DDS Structure type according to the following convention: \(<\text{StructureTypeName}>\text{VariableType}\). The DataType of the equivalent VariableType shall be \(<\text{StructureTypeName}>\text{DataType}\).

Each member shall be added as a HasComponent Reference Variable Nodes with:

- **NodeClass**—Variable.
- **BrowseName**—Name of the DDS member name. It shall match the member name used in the definition of \(<\text{StructureTypeName}>\text{DataType}\).
- **DataType**—OPC UA DataType equivalent to that of the member as specified by the mapping rules defined in this chapter. It shall match the type used in the definition of \(<\text{StructureTypeName}>\text{DataType}\).
- **TypeDefinition**—BaseDataVariableType.
- **ModelingRule**—"Optional" for DDS optional members and "Mandatory" for every other member.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>(&lt;\text{StructureTypeName}&gt;\text{VariableType})</td>
</tr>
<tr>
<td>DataType</td>
<td>(&lt;\text{StructureTypeName}&gt;\text{DataType})</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1 (for scalar Structures)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>DataType</th>
<th>TypeDefinition</th>
<th>ModelingRule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype of BaseDataVariableType.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HasComponent</td>
<td>Variable</td>
<td>(&lt;\text{MemberName}&gt;)</td>
<td>(&lt;\text{EquivalentType}&gt;)</td>
<td>BaseDataVariableType</td>
<td>Mandatory/Optional</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### 9.2.4.1.3 Example (non-normative)

Let us use **ShapeType** to illustrate the mapping of a simple structured type to OPC UA. This type is used in DDS demo applications that vendors often use to illustrate DDS concepts and test interoperability.

**ShapeType** in represented in IDL as follows:

```
struct ShapeType {
    string color;
    int32 x;
    int32 y;
    int32 shapesize;
} ;
```

To represent **ShapeType** in OPC UA we need to define an equivalent **DataType** named **ShapeTypeDataType** (i.e., the base **Structure DataType**) and an equivalent **VariableType** named **ShapeTypeVariableType** that the OPC UA Server will instantiate to represent instances of **ShapeType**.

Figure 9.13 shows the OPC UA Nodes and References involved in the mapping.
The equivalent ShapeTypeDataType is defined in Table 9.17.

Table 9.17: Example of Structure DataType Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShapeTypeDataType</td>
<td>Structure</td>
<td>This structure represents the DDS ShapeType.</td>
</tr>
<tr>
<td>color</td>
<td>String</td>
<td>Member of the structure representing the color of the shape.</td>
</tr>
<tr>
<td>x</td>
<td>Int32</td>
<td>Member of the structure representing the x position of a shape in a</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>y</td>
<td>Int32</td>
<td>Member of the structure representing the y position of a shape in a coordinate plane.</td>
</tr>
<tr>
<td>shapesize</td>
<td>Int32</td>
<td>Member of the structure representing the size of the shape.</td>
</tr>
</tbody>
</table>

The equivalent `ShapeTypeVariableType` is defined in Table 9.18.

### Table 9.18: Example of Structure VariableType Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>ShapeTypeVariableType</code></td>
</tr>
<tr>
<td>DataType</td>
<td><code>ShapeTypeDataType</code></td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
</tr>
</tbody>
</table>

Subtype of `BaseDataVariableType`.

<table>
<thead>
<tr>
<th>HasComponent</th>
<th>Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Variable</td>
<td>color</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Variable</td>
<td>x</td>
<td>Int32</td>
</tr>
<tr>
<td></td>
<td>Variable</td>
<td>y</td>
<td>Int32</td>
</tr>
<tr>
<td></td>
<td>Variable</td>
<td>shapesize</td>
<td>Int32</td>
</tr>
</tbody>
</table>

Finally, defines `my_shape`, a Variable representing an Instance of `ShapeType`.

### Table 9.19: Example of Structure Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>my_shape</code></td>
<td>Name of the <code>my_shape</code> instance of <code>ShapeType</code>.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td>The value is a scalar.</td>
</tr>
<tr>
<td>Value</td>
<td>Color = “BLUE”</td>
<td>Value indicates the current color, position, and size of the shape.</td>
</tr>
<tr>
<td></td>
<td>x = 150</td>
<td></td>
</tr>
<tr>
<td></td>
<td>y = 25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>shapesize=30</td>
<td></td>
</tr>
</tbody>
</table>

OPC UA/DDS Gateway 1.0
9.2.4.2 Union Types

9.2.4.2.1 Overview (non-normative)

In DDS, Union Types are complex types composed of a well-known discriminator member and a set of type-specific members [DDS-XTYPES].

The discriminator member—identified by the name “discriminator”—is guaranteed to be the first element of the Union and may be of the following types: Boolean, Byte, Char8, Char16, Int16, UInt16, Int32, UInt32, Int64, UInt64, Enum, and Bitmask. Alias Types resolving to those types are also valid discriminator types. The value of the discriminator may change at any moment, thereby changing the selected type-specific member.

Type-specific members may be associated with one or more values of the discriminator and may be selected because they are either associated with a specific discriminator value or they are associated with the default value.

The following example illustrates the definition of a DDS Union in IDL:

```idl
union ExampleUnion switch(int32) {
    case 1:
        int32 int32_value;
    case 2:
        int64 int64_value;
}
```

In OPC UA, Unions are standard abstract DataTypes derived from the Structure DataType [OPCUA-03]. As specified in [OPCUA-06], these structured types contain a switch field that serves a union discriminator and a set of fields that represent each of the type-specific members of the union.

The switch field is represented with a UInt32. Therefore, the maximum number of elements of the union discriminator is \(2^{32} - 1\) (the switch value 0 is reserved to indicate no fields are present, i.e., that the Union has NULL value). Switch fields of a value greater than the number of fields in the Union are invalid; thus, switch fields must be set consecutively, no gaps are allowed.

The following example, illustrates the definition of an OPC UA Union using the OPC UA Binary Schema:

```xml
<opc:StructuredType Name="ExampleUnion">
    <opc:Field Name="SwitchValue" TypeName="opc:UInt32" />
    <opc:Field Name="int32_value" TypeName="opc:Int32" SwitchField="SwitchValue" SwitchValue="1"/>
    <opc:Field Name="int64_value" TypeName="opc:Int64" SwitchField="SwitchValue" SwitchValue="2"/>
</opc:StructuredType>
```
Every DDS Union type shall be mapped to an OPC UA Union DataType. Instances of DDS Union types shall be represented as Variables of the specified DataType as shown in Figure 9.14.

Figure 9.14: Union Types Mapping to OPC UA

The Union DataType shall be defined as a subtype of the standard Union DataType. It shall be named after the original DDS Union type according to the following naming convention: `<UnionTypeName>DataType`. Every union case member of the structure shall be added as a child field where:

- The field name shall match the name of the DDS union case member, including capitalization.
- The field type shall be the OPC UA type equivalent to that of the union case member, as specified by the mapping rules defined in this chapter.
- The switch value shall be a value assigned in consecutive order—starting from 1—based on the position of the case member in the definition of the Union. Implementations of the OPC UA/DDS Gateway shall be able to map switch values to their corresponding union discriminator values and vice versa—even when different DDS union discriminator values identify the same case member. Lastly, default case members shall be treated like any other union case members; that is, they shall be assigned a switch value in the order in which they were declared.

Because in OPC UA switch fields are represented with a UInt32 value, DDS Union Types with more than $2^{32}-1$ case members (i.e., with more than 4 billion—4,294,967,295—case members) may not be represented in OPC UA and are therefore unsupported\(^{16}\).

Table 9.20 formally defines an OPC UA Union DataType equivalent to a DDS Union Type.

Table 9.20: Union Data Type Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;UnionTypeName&gt;DataType</code></td>
<td>Union</td>
<td>Union representing the DDS Union type.</td>
</tr>
<tr>
<td><code>&lt;SwitchField&gt;</code></td>
<td>UInt32</td>
<td>Switch field is the first member of the structure representing the OPC UA Union. Its type limits the</td>
</tr>
</tbody>
</table>

\(^{16}\) DDS Union Types with more than $2^{32}-1$ case members require an Int64 or UInt64 union discriminator.
number of union members to $2^{32} - 1$ fields. Thus, DDS Unions with more than $2^{32} - 1$ cases are unsupported by this specification.

Table 9.21 formally specifies an instance of a DDS Union in OPC UA using a **Variable Node**.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>&lt;UnionName&gt;</code></td>
<td>Name of the instance of the Union type the <strong>Variable</strong> represents.</td>
</tr>
<tr>
<td>DataType</td>
<td><code>&lt;UnionTypeName&gt;DataType</code></td>
<td><em>NodeId</em> of the OPC UA equivalent type representing the Union.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td><em>ValueRank</em> of -1 to indicate the <strong>Variable</strong> contains a scalar value.</td>
</tr>
</tbody>
</table>

**References**

| HasTypeDefintion   | VariableType                              | BaseDataVariable Type | Because this is a simple DataVariable with no more concrete type definition needs, it shall be defined as a BaseDataVariableType Variable. |

### 9.2.4.2.3 Example (non-normative)

Let us use the following example to illustrate the mapping of a DDS Union type instance to an OPC UA **Variable**.

An **ElementValue** Union with different case members is represented in IDL as follows:

```c
enum ElementValueType {
    INT16_VALUE,
    INT32_VALUE,
    INT64_VALUE
};

union ElementValue switch(ElementValueType) {
    case INT16_VALUE:
        int16 int16_value;
    case INT32_VALUE:
        int32 int32_value;
```
default:
  case INT64_VALUE:
    int64 int64_value;
 }

To represent `ElementValue` in OPC UA, we shall define an equivalent `DataType` named `ElementValueDataType`. Instances of `ElementValue`, such as `ElementValue my_value`, shall be represented as OPC UA `Variables` of `ElementValueDataType`.

Figure 9.15 shows the OPC UA `Nodes` and `References` involved in the mapping.

![Figure 9.15: Example of Union Type Mapping to OPC UA](image)

Table 9.22 shows the equivalent `ElementValueDataType`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Switch Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElementValueDataType</td>
<td>Union</td>
<td>N/A</td>
<td>This Union represents the DDS <code>ElementValue</code> union.</td>
</tr>
<tr>
<td>int16_value</td>
<td>Int16</td>
<td>1</td>
<td>Case member for <code>INT16_VALUE</code>.</td>
</tr>
<tr>
<td>int32_value</td>
<td>Int32</td>
<td>2</td>
<td>Case member for <code>INT32_VALUE</code>.</td>
</tr>
<tr>
<td>int64_value</td>
<td>Int64</td>
<td>3</td>
<td>Case member for both <code>INT64_VALUE</code> and <code>default</code>.</td>
</tr>
</tbody>
</table>

Table 9.23 defines `my_value`, a `Variable` representing an Instance of `ElementValue`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>my_value</td>
<td>Name of the <code>my_value</code> instance of the <code>ElementValueDataType</code>.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td>The value is a scalar.</td>
</tr>
<tr>
<td>Value</td>
<td>Switch Field = 2</td>
<td>The switch field of the Union is 2. Therefore, in this case</td>
</tr>
</tbody>
</table>

Implementers of the OPC UA/DDS Gateway must keep track of both discriminator values internally.
int32_value = 4  ElementValueDataType is providing an int32_value that is equal to 4.

<table>
<thead>
<tr>
<th><strong>DataType</strong></th>
<th><strong>ElementValueDataType</strong></th>
<th><strong>NodeId</strong> of <strong>ElementValueDataType</strong>.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDef</td>
<td>VariableType</td>
<td>BaseDataVariableType</td>
<td>Because this is a simple DataVariable with no more concrete definition needs, it shall be defined as a BaseDataVariableType.</td>
</tr>
</tbody>
</table>

### 9.2.5 Collection Types

Collection types represent containers for elements of homogeneous types [DDS-XTYPES]. The DDS Type System defines three types of containers: Arrays, Sequences, and Maps.

#### 9.2.5.1 Arrays

**9.2.5.1.1 Overview (non-normative)**

Arrays are fixed-size one- or multi-dimensional collections. That is, all instances of a given array type shall have the same number of elements of a certain type.

**9.2.5.1.2 Mapping**

**9.2.5.1.2.1 Arrays of Primitive and String Types**

Arrays of Primitive and String types shall be mapped to Variables of the corresponding OPC UA built-in type as shown in Figure 9.16. These Variable Nodes may become part of complex VariableTypes or ObjectTypes as a result of the mappings defined in this specification.
Table 9.24 formally specifies the representation of an Array of Primitive or String types in OPC UA using a *Variable* Node.

**Table 9.24: Array of Primitive or String Type Variable Definition**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td><em>BrowseName</em> shall be a string matching the name of the DDS variable with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>&lt;UInt32&gt; &gt;= 1</td>
<td><em>ValueRank</em> shall be equal to the number of dimensions of the DDS array. For example, if the array has two dimensions, <em>ValueRank</em> shall be 2.</td>
</tr>
<tr>
<td>ArrayDimensions</td>
<td>&lt;UInt32[]&gt;</td>
<td><em>ArrayDimensions</em> array shall have a number of elements equal to the number of dimensions of the DDS array (i.e., equal to <em>ValueRank</em>). Each element of the <em>ArrayDimensions</em> array shall specify the size of the corresponding dimension in the original DDS Array type. For example, if a DDS array has two dimensions of size 32 and 64, respectively; <em>ArrayDimensions</em> shall be [32, 64].</td>
</tr>
</tbody>
</table>
**DataType** | `<NodeId>` | **DataType** shall point to the **NodeId** of the OPC UA type equivalent to that of the array elements.

- If the array is of a DDS Primitive type, **DataType** shall point to the **NodeId** of the equivalent type according to the rules specified in Table 9.2.
- If the array is of a String type, **DataType** shall point to the **NodeId** of the equivalent OPC UA built-in type specified in sub clause 9.2.2.2 (see Table 9.4 for **String8** Types and Table 9.5 for **String16** types).

**References** | **NodeClass** | **BrowseName** | **Description**
--- | --- | --- | ---
HasTypeDefinition | VariableType | BaseDataVariableType | Because this is a simple **DataVariable** with no more concrete type definition needs, it shall be defined as a **BaseDataVariableType** Variable.

### 9.2.5.1.2.2 Arrays of Enumerated Types

Arrays of Enumerated types shall be mapped to OPC UA **Variables** of the corresponding **Enumeration** or **OrderedSet** **DataType**. These **Variable Nodes** may become part of complex **VariableTypes** or **ObjectTypes** as a result of the mappings defined in this specification.

Figure 9.17 shows the **Nodes** and **References** involved in the mapping of an Array of Enumerations to OPC UA.

![Figure 9.17: Array of Enumerations Mapping to OPC UA](image.png)

Table 9.25 formally specifies the representation of an Array of Enumerations in OPC UA using a **Variable Node**.

### Table 9.25: Array of Enumerations Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>&lt;String&gt;</code></td>
<td><strong>BrowseName</strong> shall be a string matching the name of the DDS variable representing the Array of Enumerations with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td><code>&lt;UInt32</code> &gt;= 1</td>
<td><strong>ValueRank</strong> shall be equal to the number of dimensions of the DDS array. For example, if the array has two dimensions, <strong>ValueRank</strong> shall be 2.</td>
</tr>
</tbody>
</table>
**ArrayDimensions** | `<UInt32[]>` | *ArrayDimensions* array shall have a number of elements equal to the number of dimensions of the DDS array (i.e., equal to *ValueRank*). Each element of the *ArrayDimensions* array shall specify the size of the corresponding dimension in the original DDS Array Type.

For example, if a DDS array has two dimensions of size 32 and 64, respectively; *ArrayDimensions* shall be `[32, 64]`.

**DataType** | `<NodeId>` | *DataType* shall point to the *NodeId* of `<EnumerationTypeName>DataType` (as specified in Table 9.7). Variables representing scalar Enumerations and Arrays of Enumerations share the same *DataType*.

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDefinition</td>
<td>VariableType</td>
<td>BrowseName = BaseDataVariable Type</td>
<td>Because this is a simple DataVariable with no more concrete definition needs, it shall be defined as a BaseDataVariableType Variable.</td>
</tr>
</tbody>
</table>

Figure 9.18 shows the *Nodes* and *References* involved in the mapping of an Array of Bitmasks to OPC UA.

**Figure 9.18: Array of Bitmasks Mapping to OPC UA**

Table 9.26 formally specifies the representation of an Array of Bitmasks in OPC UA using a Variable Node.

**Table 9.26: Array of Bitmasks Variable Definition**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>&lt;String&gt;</code></td>
<td><em>BrowseName</em> shall be a string matching the name of the DDS variable representing the Array of Bitmasks with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td><code>&lt;UInt32&gt;</code>  &gt;= 1</td>
<td><em>ValueRank</em> shall be equal to the number of dimensions of the DDS array. For example, if the array has two dimensions, <em>ValueRank</em> shall be 2.</td>
</tr>
</tbody>
</table>
**ArrayDimensions** | `<UInt32[]>` | *ArrayDimensions* array shall have a number of elements equal to the number of dimensions of the DDS array (i.e., equal to *ValueRank*). Each element of the *ArrayDimensions* array shall specify the size of the corresponding dimension in the original DDS Array Type.

For example, if a DDS array has two dimensions of size 32 and 64, respectively; *ArrayDimensions* shall be [32, 64].

**DataType** | `<NodeId>` | *DataType* shall point to the NodeId of `<BitmaskTypeName>DataType` (as specified in Table 9.13). Variables representing scalar Bitmasks and Arrays of Bitmasks share the same *DataType*.

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDefinition</td>
<td>VariableType</td>
<td>BaseDataVariable Type</td>
<td>Because this is a simple <em>DataVariable</em> with no more concrete definition needs, it shall be defined as a <em>BaseDataVariableType</em> Variable</td>
</tr>
</tbody>
</table>

### 9.2.5.1.2.3 Arrays of Structures

Arrays of Structures shall be mapped to OPC UA *Variable Nodes* representing fixed-size one- or multi-dimensional arrays of the equivalent *Structure DataType*. These *Variable Nodes* may become part of complex *VariableTypes* or *ObjectTypes* as a result of the mappings defined in this specification.

Figure 9.19 shows the *Nodes* and *References* involved in the mapping of an Array of Structures to OPC UA.
Variable Nodes representing an Array of Structures shall be constructed as follows:\(^{18}\):

- The `DataType` of the `Variable` shall be the equivalent OPC UA `Structure DataType` specified in Table 9.15. Thus, `Variables` representing scalar Structures and Arrays of Structures share the same `DataType`: `<StructureTypeName>DataType`.

- The `Value` `Attribute` of the `Variable` shall be capable of storing a fix-length one- or multi-dimensional arrays of `<StructureTypeName>DataType`. Therefore, the `Variable` shall be instantiated with `ValueRank` 1 or more and `ArrayDimensions` with the equivalent number of elements specifying each dimension’s length. This configuration enables OPC UA Clients capable of deserializing `<StructureTypeName>DataType` to read the whole array in one operation.

- The `Variable` shall define a set of `HasOrderedComponent` `References` to `Variables` representing each element of the array. These `Variables` shall be defined as instances of `<StructureTypeName>VariableType` (as specified in Table 9.16) and shall be named according to the following convention: `<StructureTypeName>VariableType` ; where `<StructureTypeName>` is the name of the original DDS Structure type and `<index>` is the position of the element in the array. If the array is multi-dimensional `<index>` will represent the position in each dimension separated by underscores (e.g., for position [1][2][3] `<index>` will be 1_2_3 and the Structure’s name `<StructureTypeName>_1_2_3`). This mapping enables generic OPC UA Clients incapable of deserializing `<StructureTypeName>DataType` to process every element of the `Array` by recursively following the

---

\(^{18}\) This mapping is based on the guidelines for modeling arrays of complex variables defined in clause A.6 of [OPCUA-05].
HasComponent References specified by `<StructureTypeName>VariableType` to provide separate access to the Structure members.

Table 9.27 formally specifies the representation of an Array of Structures in OPC UA using a Variable Node.

### Table 9.27: Array of Structures Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>&lt;String&gt;</code></td>
<td><code>BrowseName</code> shall be a string matching the name of the DDS variable representing the Array of Structures with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td><code>&lt;UInt32&gt; &gt;= 1</code></td>
<td><code>ValueRank</code> shall be equal to the number of dimensions of the DDS array. For example, if the array has two dimensions, <code>ValueRank</code> shall be 2.</td>
</tr>
<tr>
<td>ArrayDimensions</td>
<td><code>&lt;UInt32[]&gt;</code></td>
<td><code>ArrayDimensions</code> array shall have a number of elements equal to the number of dimensions of the DDS array (i.e., equal to <code>ValueRank</code>). Each element of the <code>ArrayDimensions</code> array shall specify the size of the corresponding dimension in the original DDS Array Type. For example, if a DDS array has two dimensions of size 32 and 64, respectively; <code>ArrayDimensions</code> shall be <code>[32, 64]</code>.</td>
</tr>
<tr>
<td>DataType</td>
<td><code>&lt;NodeID&gt;</code></td>
<td><code>DataType</code> shall point to the NodeId of <code>&lt;StructureTypeName&gt;DataType</code> (as specified in Table 9.15). Variables representing scalar Structures and Arrays of Structures share the same <code>DataType</code>.</td>
</tr>
</tbody>
</table>

### References

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>DataType</th>
<th>TypeDefinition</th>
<th>ModelingRule</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDefinition</td>
<td>VariableType</td>
<td>BaseDataVariableType</td>
<td>BaseDataType</td>
<td>BaseVariableType</td>
<td>Mandatory</td>
</tr>
<tr>
<td>HasOrderedComponent</td>
<td>Variable</td>
<td><code>&lt;StructureTypeName&gt;_&lt;index&gt;</code></td>
<td><code>&lt;StructureTypeName&gt;DataType</code></td>
<td><code>&lt;StructureTypeName&gt;VariableType</code> (as specified in Table 9.16)</td>
<td>Mandatory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 9.2.5.1.2.4 Arrays of Union Types

Arrays of Unions shall be mapped to OPC UA Variables of the corresponding Union `DataType`. These Variable Nodes may become part of complex VariableTypes or ObjectType as a result of the mappings defined in this specification.
Figure 9.20 shows the **Nodes and References** involved in the mapping of an Array of Unions to OPC UA.

![Figure 9.20: Array of Unions Mapping to OPC UA](image)

Table 9.28 formally specifies the representation of an Array of Union types in OPC UA using a **Variable Node**.

**Table 9.28: Array of Unions Variable Definition**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td><code>BrowseName</code> shall be a string matching the name of the DDS variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>representing the Array of Unions with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>&lt;UInt32&gt; &gt;= 1</td>
<td><code>ValueRank</code> shall be equal to the number of dimensions of the DDS Array.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, if the array has two dimensions, <code>ValueRank</code> shall be 2.</td>
</tr>
<tr>
<td>ArrayDimensions</td>
<td>&lt;UInt32[]&gt;</td>
<td>The <code>ArrayDimensions</code> array shall have a number of elements equal to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number of dimensions of the DDS Array (i.e., equal to <code>ValueRank</code>). Each</td>
</tr>
<tr>
<td></td>
<td></td>
<td>element of the <code>ArrayDimensions</code> array shall specify the size of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>corresponding dimension in the original DDS Array type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, if a DDS Array has two dimensions of size 32 and 64, respectively; <code>ArrayDimensions</code> shall be [32, 64].</td>
</tr>
<tr>
<td>DataType</td>
<td>&lt;NodeId&gt;</td>
<td><code>DataType</code> shall point to the NodeId of <code>&lt;UnionTypeName&gt;DataType</code> (as</td>
</tr>
<tr>
<td></td>
<td></td>
<td>specified in Table 9.20). Variables representing scalar Unions and Arrays of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unions share the same <code>DataType</code>.</td>
</tr>
</tbody>
</table>

**References**

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDefinition</td>
<td>VariableType</td>
<td>BaseDataVariableType</td>
<td>Because this is a simple DataVariable with no more concrete definition needs, it shall be defined as a BaseDataVariableType Variable.</td>
</tr>
</tbody>
</table>
9.2.5.1.2.5 Arrays of Collection Types

Arrays of Collection Types shall be mapped to Object Nodes with HasOrderedComponent References to Variables or Objects representing instances of the associated Collection Type as shown in Figure 9.21. These Objects may become part of complex VariableTypes or ObjectTypes as a result of the mappings specified in this chapter.

Table 9.29: Array of Collection Types Object Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;ArrayVariableName&gt;</td>
<td>Name of the instance of an Array of Collection Types the Object represents.</td>
</tr>
<tr>
<td>IsAbstract</td>
<td>False</td>
<td>Objects representing an Array of Collection Types are never abstract.</td>
</tr>
<tr>
<td>References</td>
<td>NodeClass</td>
<td>BrowseName</td>
</tr>
<tr>
<td></td>
<td>Subtype of BaseObjectType.</td>
<td>&lt;ArrayVariableName&gt;_&lt;index&gt;</td>
</tr>
<tr>
<td>HasOrderedComponent</td>
<td>Variable or Object</td>
<td>&lt;ArrayVariableName&gt;_&lt;index&gt;</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 9.30 defines the structure of a Variable or Object Node representing a Collection within the Array of Collections.

Table 9.30: Collection Variable or Object Definition – Arrays of Collections

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;ArrayName&gt;_&lt;index&gt;</td>
<td>The BrowseName is composed of the &lt;ArrayVariableName&gt; and an &lt;index&gt; suffix indicating the position of the Collection element in the Array.</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>Attributes of the Variable or Object Node representing the Collection.</td>
</tr>
</tbody>
</table>
9.2.5.1.3 Example (non-normative)

Let us use the following example to illustrate the mapping of an Array type to an OPC UA Variable.

An array of 32-bit integers, member of a Structure type, is represented in IDL as follows:

```cpp
struct StructuredType {
    int32 my_array[4];
};
```

To represent `my_array` in OPC UA, we shall create a `Variable` following the rules specified in Table 9.24. Table 9.31 shows the definition of this Variable.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>my_array</td>
<td><code>BrowseName</code> matches the name of the original DDS variable: <code>my_array</code>.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>1</td>
<td><code>ValueRank</code> of 1 to indicate the <code>Variable</code> contains a one-dimensional array.</td>
</tr>
<tr>
<td>ArrayDimensions</td>
<td>[4]</td>
<td><code>ArrayDimensions</code> has a single element with value 4 to indicate the array has one dimension with 4 elements.</td>
</tr>
<tr>
<td>DataType</td>
<td>Int32</td>
<td><code>NodeId</code> of <code>Int32</code>, the equivalent type for the elements of the DDS Array.</td>
</tr>
<tr>
<td>Value</td>
<td>[&lt;Int32&gt;, &lt;Int32&gt;, &lt;Int32&gt;, &lt;Int32&gt;]</td>
<td>A valid Array, containing four 32-bit integer values. When the <code>Variable</code> is used in the definition of a complex <code>VariableType</code> or <code>ObjectType</code>, <code>Value</code> may be overwritten by the instance of the corresponding <code>Instance Type</code>.</td>
</tr>
</tbody>
</table>

9.2.5.2 Sequences

9.2.5.2.1 Overview (non-normative)

Sequence types are variable-size one-dimensional collections. That is, different instances of a given sequence type may have a different number of elements of a certain type. Sequences may be defined as bounded or unbounded, depending on whether the maximum number of elements that the sequence may contain is specified.

9.2.5.2.2 Mapping

9.2.5.2.2.1 Sequences of Primitive and String Types

Sequences of Primitive and String types shall be mapped to `Variables` of the corresponding OPC UA built-in type as show in Figure 9.22. These `Variable Nodes` may become part of complex `VariableTypes` or `ObjectTypes` as a result of the mappings defined in this specification.
Table 9.32 formally specifies the representation of a Sequence of Primitive or String types in OPC UA using a Variable Node.

Table 9.32: Sequence of Primitive or String Types Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td>*BrowseName shall be a string matching the name of the DDS variable with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>1</td>
<td>*ValueRank shall be 1, indicating that the sequence has one dimension.</td>
</tr>
<tr>
<td>ArrayDimensions</td>
<td>[0]</td>
<td>*ArrayDimensions array shall include a single element of value 0 (which indicates the only dimension has variable length).</td>
</tr>
<tr>
<td>DataType</td>
<td>&lt;NodeId&gt;</td>
<td>*DataType shall point to the NodeId of the OPC UA type equivalent to that of the sequence elements:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the sequence is of a DDS Primitive type, DataType shall point to the NodeId of the equivalent type as specified in Table 9.2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the sequence is of a String type, DataType shall point to the NodeId of the equivalent OPC UA built-in type as specified in sub clause 9.2.2.2.2 (see Table 9.4 for String8 Types and Table 9.5 for ...)</td>
</tr>
</tbody>
</table>
9.2.5.2.2 Sequences of Enumerated Types

Sequences of Enumerated types shall be mapped to OPC UA Variables of the corresponding Enumeration or OrderedSet DataType. These Variable Nodes may become part of complex VariableTypes or ObjectTypes as a result of the mappings defined in this specification.

Figure 9.23 shows the Nodes and References involved in the mapping of Sequences of Enumerations to OPC UA.

Table 9.33: Sequence of Enumerations Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td>BrowseName shall be a string matching the name of the DDS variable representing the Sequence of Enumerations with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>1</td>
<td>ValueRank shall be 1, indicating that the sequence has one dimension.</td>
</tr>
<tr>
<td>ArrayDimensions</td>
<td>[0]</td>
<td>Sequences are one-dimensional arrays of variable length. Thus, the ArrayDimensions array shall include a single element of value 0 (which indicates the only dimension has variable length).</td>
</tr>
<tr>
<td>DataType</td>
<td>&lt;NodeId&gt;</td>
<td>DataType shall point to the NodeId of &lt;EnumerationTypeName&gt;DataType. Variables representing scalar Enumerations and Sequences of Enumerations share the same DataType.</td>
</tr>
</tbody>
</table>

References

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDefinition</td>
<td>VariableType</td>
<td>BaseDataVariable Type</td>
<td>Because this is a simple DataVariable with no more concrete definition needs, it shall be defined as a BaseDataVariableType Variable.</td>
</tr>
</tbody>
</table>
Figure 9.24 shows the *Nodes* and *References* involved in the mapping of a Sequences of Bitmasks to OPC UA.

![Diagram of SequenceOfBitmasks]

Table 9.34 formally specifies the representation of a Sequence of Bitmasks in OPC UA as a *Variable Node*.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td><em>BrowseName</em> shall be a string matching the name of the DDS variable representing the Sequence of Bitmasks with the same capitalization.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>1</td>
<td><em>ValueRank</em> shall be 1, indicating that the sequence has one dimension.</td>
</tr>
<tr>
<td>ArrayDimensions</td>
<td>[0]</td>
<td>Sequences are one-dimensional arrays of variable length. Thus, the <em>ArrayDimensions</em> array shall include a single element of value 0 (which</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates the only dimension has variable length).</td>
</tr>
<tr>
<td>DataType</td>
<td>&lt;NodeId&gt;</td>
<td><em>DataType</em> shall point to the <em>NodeId</em> of <code>&lt;BitmaskTypeName&gt;DataType</code> (as specified in Table 9.13). <em>Variables</em> representing scalar Bitmasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and Sequences of Bitmasks share the same <em>DataType</em>.</td>
</tr>
</tbody>
</table>

**References**

<table>
<thead>
<tr>
<th>Reference</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDefinition</td>
<td>VariableType</td>
<td>BaseDataVariableType</td>
<td>Because this is a simple <em>DataVariable</em> with no more concrete definition needs, it shall be defined as a <em>BaseDataVariableType Variable</em>.</td>
</tr>
</tbody>
</table>

### 9.2.5.2.2.3 Sequences of Structures

Sequence of Structures shall be mapped to OPC UA *Variable Nodes* representing variable-length one-dimensional arrays of the equivalent *Structure DataType*. These *Variable Nodes* may become part of complex *VariableTypes* or *ObjectTypes* as a result of the mappings specified in this chapter.
Figure 9.25 shows the Nodes and References involved in the mapping of a Sequence of Structure types to OPC UA.

Variable Nodes representing a Sequence of Structures shall be constructed as follows:

- The **DataType** of the **Variable** shall be the equivalent OPC UA Structure Type specified in Table 9.15. Thus, Variables representing scalar Structures and Sequences of Structures share the same **DataType**: `<StructureTypeName>DataType`.

- The **Value Attribute** of the **Variable** shall be capable of storing a variable-length one-dimensional array of `<StructureTypeName>DataType`. Therefore, the **Variable** shall be instantiated with **ValueRank** equal to 1, and **ArrayDimensions** equal to [0]. This configuration enables OPC UA Clients capable of deserializing `<StructureTypeName>DataType` to read the whole array in one operation.

- The **Variable** shall define a set of **HasOrderedComponent References** to **Variables** representing each element of the sequence. These **Variables** shall be defined as instances of `<StructureTypeName>VariableType` (as specified in Table 9.16) and shall be named according to the following convention: `<StructureTypeName>_index`; where `<StructureTypeName>` is the name of the original Structure Type and `index` is the position of the element in the sequence. This mapping enables generic OPC UA Clients incapable of deserializing `<StructureTypeName>DataType` to process every element of the Sequence by recursively following the **HasComponent References** specified by `<StructureTypeName>VariableType` to provide separate access to the Structure members.

Table 9.35 formally specifies the representation of a Sequence of Structures in OPC UA using a Variable Node.

**Table 9.35: Sequence of Structures Variable Definition**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPC UA/DDS Gateway 1.0</td>
<td>91</td>
<td></td>
</tr>
</tbody>
</table>
BrowseName shall be a string matching the name of the DDS variable representing the Sequence of Structure types with the same capitalization.

ValueRank shall be 1, indicating that the sequence has one dimension.

Sequences are one-dimensional arrays of variable length. Thus, the ArrayDimensions array shall include a single element of value 0 (which indicates the only dimension has variable length).

DataType shall point to the NodeId of `<StructureTypeName>DataType` (as specified in Table 9.15). Variables representing scalar Structures and Sequences of Structures share the same DataType.

### References

<table>
<thead>
<tr>
<th>HasTypeDefinitio n</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>DataType</th>
<th>TypeDefinition</th>
<th>ModelingRule</th>
</tr>
</thead>
<tbody>
<tr>
<td>VariableType</td>
<td></td>
<td>BaseDataVariable Type</td>
<td>BaseData Type</td>
<td>BaseDataType</td>
<td>Mandatory</td>
</tr>
<tr>
<td>HasOrderedComp onent</td>
<td>Variable &lt;StructureTypeName&gt;_&lt;index&gt;</td>
<td>&lt;StructureTypeName&gt;DataType</td>
<td>&lt;StructureTypeName&gt;V ariableType (as specified in Table 9.16)</td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

#### 9.2.5.2.2.4 Sequences of Unions

Sequences of Unions shall be mapped to OPC UA Variables of the corresponding Union Type. These Variable Nodes may become part of complex VariableTypes or ObjectTypes as a result of the mappings defined in this specification.

Figure 9.26 shows the Nodes and References involved in the mapping of Sequences of Unions to OPC UA.

![Figure 9.26: Sequence of Unions Mapping to OPC UA](image)

Table 9.36 formally specifies the representation of a Sequence of Union types in OPC UA as a Variable Node.

**Table 9.36: Sequence of Unions Variable Definition**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
BrowseName shall be a string matching the name of the DDS variable representing the Sequence of Unions with the same capitalization.

ValueRank shall be 1, indicating that the sequence has one dimension.

Sequences are one-dimensional arrays of variable length. Thus, the ArrayDimensions array shall include a single element of value 0 (which indicates the only dimension has variable length).

DataType shall point to the NodeId of `<UnionTypeName>DataType`. Variables representing scalar Unions and Sequences of Unions share the same DataType.

References

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>&lt;SequenceName&gt;</code></td>
<td>Name of the instance a Sequence of Collection Types the Object represents.</td>
</tr>
<tr>
<td>IsAbstract</td>
<td>False</td>
<td>Objects representing Sequence of Collection Types are never abstract.</td>
</tr>
</tbody>
</table>

Table 9.37: Sequence of Collection Types Object Definition

Objects representing Sequences of Collection Types may refer to a set of either Variables Nodes or Object Nodes. The NodeClass and TypeDefinition of these Nodes depend on the representation of the instances of the specific collection type the Sequence contains in OPC UA.

Because Sequences are a homogeneous collections, all components of an Object Node representing a Sequence of Collection Types shall have the same NodeClass and TypeDefinition. Therefore, only one of the associations in this diagram may be instantiated at the same time.

Figure 9.27: Sequence of Collection Types Mapping to OPC UA

Table 9.37 formally specifies the representation of a Sequence of Collection Types in OPC UA using an Object Node.
Table 9.38 defines the structure of a Variable or Object Node representing a specific Collection within the Sequence of Collections.

Table 9.38: Collection Variable or Object Definition – Sequences of Collections

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td><code>&lt;SequenceName&gt;_&lt;index&gt;</code></td>
<td><code>BrowseName</code> is composed of the <code>&lt;SequenceName&gt;</code> and an <code>&lt;index&gt;</code> indicating the position of the Collection element in the Sequence.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>Attributes of the Variable or Object Node representing the Collection.</td>
</tr>
<tr>
<td>References</td>
<td>NodeClass</td>
<td>BrowserName</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>DataTypes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>TypeDefinitions</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>ModelingRules</td>
</tr>
</tbody>
</table>

9.2.5.2.3 Example (non-normative)

Let us use the following example to illustrate the mapping of a Sequence type to an OPC UA Variable.

An unbounded Sequence of 32-bit integers, member of a Structure type, is represented in IDL as follows:

```idl
struct StructuredType {
    sequence<int32> my_sequence;
};
```

To represent `my_sequence`, we shall create a Variable following the rules specified in Table 9.32. Table 9.39 shows the definition of this Variable.

Table 9.39: Example of Sequence Variable Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>my_sequence</td>
<td><code>BrowseName</code> matches the name of the original DDS variable: <code>my_sequence</code>.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>1</td>
<td><code>ValueRank</code> of 1 to indicate the <code>Variable</code> contains a one-dimensional array (i.e., a sequence).</td>
</tr>
<tr>
<td>ArrayDimensions</td>
<td>[0]</td>
<td><code>ArrayDimensions</code> has a single element with value zero to indicate that the only dimension has variable length.</td>
</tr>
<tr>
<td>DataType</td>
<td>Int32</td>
<td><code>NodeId</code> of Int32, the equivalent type for the elements of the DDS Sequence.</td>
</tr>
<tr>
<td>Value</td>
<td><code>[&lt;Int32&gt;, &lt;Int32&gt;, &lt;Int32&gt;]</code></td>
<td>A valid Sequence, containing a three 32-bit integer values. When the <code>Variable</code> is used in the definition of a complex <code>VariableType</code> or <code>ObjectType</code>, <code>Value</code> may be overwritten by the instance of the corresponding Instance type.</td>
</tr>
</tbody>
</table>

References

94 OPC UA/DDS Gateway 1.0
9.2.5.3 Maps

9.2.5.3.1 Overview (non-normative)
Maps are variable-size associative collections. They provide a simple way of organizing a homogeneous collection of elements by an associated key. In practice, a map can be seen as sequence of structured types containing a key-value pair. The IDL representation of such implementation would be the following:

```
struct MapEntry {
    <KeyType> key;
    <ValueType> value;
};
sequence<MapEntry> MapType; // sequence<MapEntry, <bound>> for bounded maps
```

With this approach, an application seeking to retrieve a certain value must first search for the appropriate key value in the sequence of map entries, and then access the value member of the structure.

[DDS-XTYPES] specifies in sub clause 7.2.2.4.3 that “implementers (...) need only support key elements of signed and unsigned integer types and of narrow and wide string types” and “the behavior of maps with other key element types is undefined and may not be portable.” As a result, this specification only addresses the mapping of Map types with integer and string key types.

9.2.5.3.2 Mapping
Maps shall be represented as Object Nodes with HasComponent References to Variable or Object Nodes representing the associated MapEntries as shown in Figure 9.28. Map Objects may become part of complex VariableTypes or ObjectTypes as a result of the mappings defined in this specification.

MapEntry Nodes shall be modeled according to the mapping rules specified in this chapter for its value element type (i.e., <ValueType> in the MapEntry definition of sub clause 9.2.5.3.1). Because those mapping rules associate instances of DDS types to either Objects or Variables depending on the type, a MapEntry may be represented as an Object or a Variable Node.
The BrowseName of each map MapEntry shall be the string representation of its key element (i.e., the string representation of the specific instance of <KeyType> in the MapEntry definition of sub clause 9.2.5.3.1).

Table 9.40 defines the structure Object Nodes representing instances of a DDS Map.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;-MapName&gt;</td>
<td>Name of the instance of Map type the Object represents.</td>
</tr>
<tr>
<td>IsAbstract</td>
<td>False</td>
<td>Objects representing DDS Maps are never abstract.</td>
</tr>
</tbody>
</table>

Table 9.40: Map Object Definition

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>DataType</th>
<th>TypeDefinition</th>
<th>ModelingRule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype of the BaseObjectType.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HasComponent</th>
<th>Variable or Object</th>
<th>&lt;MapEntryKeyIdStringRepresentation&gt;</th>
<th>&lt;NodeId&gt;</th>
<th>&lt;MapEntryValueEquivalentTypeDef&gt;</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 9.41 defines the structure of a Variable or Object Node representing a specific MapEntry.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;-MapEntryKeyIdStringRepresentation&gt;</td>
<td>String representation of the key element of the MapEntry.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>DataType</th>
<th>TypeDefinition</th>
<th>ModelingRule</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

9.2.5.3.3 Example (non-normative)

Let us use the following example to illustrate the mapping of a Map type to an OPC UA Object.

A Map with String keys and Int32 values, member of a Structure type, is represented in IDL as follows:

```idl
define StructuredType {
    map<string, int32> my_map;
}
```

Let us also assume that `my_map` has been instantiated and contains two MapEntries:

```cpp
my_map["Manuela"] = 57;
my_map["JoseMaria"] = 51;
```

As specified above, to represent `my_map` we need to:

1. Create two Nodes of Variable or Object NodeClass to represent the two existing MapEntries (see Table 9.41).

Since in this case the value element type of `my_map` is Int32, MapEntries shall be represented as OPC UA...
Variables of DataType Int32 (see sub clause 9.2.1.2). The BrowseName of each Variable shall be the string representation of each MapEntry’s key element; i.e.: “Manuela” and “JoseMaria”.

2. Instantiate an Object Node to represent the Map (see Table 9.40).

Figure 9.29 shows the OPC UA Nodes and References involved in the mapping.

Table 9.42 and Table 9.43 show the definition for the Variables representing the different MapEntries in my_map:

Table 9.42: Example of MapEntry Variable Definition – First MapEntry

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>Manuela</td>
<td><em>BrowseName</em> is the string representation of the key element of the MapEntry.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td><em>ValueRank</em> of -1 to indicate the Variable contains a scalar value.</td>
</tr>
<tr>
<td>DataType</td>
<td>Int32</td>
<td><em>NodeId</em> of Int32, the type equivalent to a DDS 32-bit integer (which is the type of the value element of the MapEntry).</td>
</tr>
<tr>
<td>Value</td>
<td>57</td>
<td>Value of the MapEntry.</td>
</tr>
</tbody>
</table>

References

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDefinition</td>
<td>VariableType</td>
<td>BaseDataVariable Type</td>
<td>Because this is a simple DataVariable with no more concrete type definition needs, it shall be defined as a</td>
</tr>
</tbody>
</table>
Table 9.43: Example of MapEntry Variable Definition – Second MapEntry

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>JoseMaria</td>
<td><em>BrowseName</em> is the string representation of the key element of the <em>MapEntry.</em></td>
</tr>
<tr>
<td>ValueRank</td>
<td>-1</td>
<td><em>ValueRank</em> of -1 to indicate the <em>Variable</em> contains a scalar value.</td>
</tr>
<tr>
<td>DataType</td>
<td>Int32</td>
<td><em>NodeId</em> of Int32, the type equivalent to a DDS 32-bit integer (which is the type of the value element of the <em>MapEntry</em>).</td>
</tr>
<tr>
<td>Value</td>
<td>51</td>
<td>Value of the <em>MapEntry</em>.</td>
</tr>
</tbody>
</table>

Table 9.44 shows the definition of the *Object Node* representing *my_map*.

Table 9.44: Example of Map Object Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>my_map</td>
<td>Name of the instance of a Map this <em>Object</em> represents.</td>
</tr>
<tr>
<td>IsAbstract</td>
<td>False</td>
<td>This <em>Object</em> is not abstract.</td>
</tr>
</tbody>
</table>

| HasTypeDefinition | ObjectType | BaseObjectType | Because this is a simple *Object* with no more concrete type definition needs, it shall be defined as an *Object* of *BaseObjectType*. |

<table>
<thead>
<tr>
<th>HasComponent</th>
<th>Variable</th>
<th>Reference to one of the <em>MapEntries</em>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>JoseMaria</td>
<td>Variable</td>
<td>Reference to one of the <em>MapEntries</em>.</td>
</tr>
</tbody>
</table>

9.2.6 Nested Types

9.2.6.1 Overview (non-normative)

Nested Types are data types that appear only as members of other types. In IDL, these are documented with the *@nested* annotation, which indicates the IDL compiler that no *DataWriter*, *DataReader*, or *TypeSupport* classes shall be generated for the annotated types.
9.2.6.2 Mapping

Implementations of this specification generating DataWriter, DataReader, or TypeSupport classes based on type representation languages supporting the @nested (e.g., IDL and XML) shall not generate such classes for types marked as nested either.

Other than that, types marked as @nested shall be mapped according to the general mapping rules specified in this chapter.

9.2.7 Alias Types

9.2.7.1 Overview (non-normative)

Alias types—also referred to as typedefs from their representation in IDL—introduce an additional name for an existing type. The purpose of Alias types is to provide a more human-readable name to help understand the semantics and uses of a given type.

9.2.7.2 Mapping

The alternative name specified by the Alias types shall be ignored when mapping DDS types to OPC UA. That is, Alias types and instances of Alias types shall be mapped as if the alternative type name were the original type name.

9.2.7.3 Example (non-normative)

An array of Entero32—an alias of Int32—represented in IDL as follows:

```idl
typedef int32 Entero32;
sequence<Entero32> my_sequence;
```

Shall be mapped, as specified in 9.2.5.2, to the OPC UA Variable described in Table 9.39. That is, it shall be mapped as my_sequence were simply defined as a sequence of int32:

```idl
sequence<int32> my_sequence;
```

9.2.8 Keyed Types

As specified in [DDS-XTYPES], structure members and union discriminators can be marked as key members. These members determine the Instance of a Topic a data sample belongs to.

To enable the Instance creation lifecycle specified in sub clause 9.3.4.6:

- The WriteMask Attribute of Variable Nodes representing key members of a structures shall be undefined (i.e., set to 0).
- The union discriminator is not directly exposed in the AddressSpace of the OPC UA Server; therefore, a mapping for key union discriminators is unnecessary.

9.3 DDS Global Data Space Mapping

This clause defines a complete mapping of the DDS Global Data Space to OPC UA.
9.3.1 Overview (non-normative)

9.3.1.1 DDS Global Data Space and DDS

As explained in clause 7.2.1, the DDS data model defines a logical Global Data Space where Publisher and Subscriber applications send and receive data objects.

The DDS Global Data Space is divided into different logical portions named Domains. A Domain establishes a virtual network that links all the applications that share the same DomainId; therefore, it isolates DDS applications from applications running on different Domains [DDS].

DDS applications exchange data objects in the form of Topics, which have an associated type. Topics may have different Instances, which are identified by a key built upon all the key members of its type. If no key is provided, the data set associated with a Topic is restricted to a single instance [DDS].

To provide applications with the necessary means to participate in the Global Data Space and perform operations in it DDS defines a complete set of Entities:

- DomainParticipants allow applications to join a certain Domain; create Topics, Publishers and Subscribers; and register types.
- Publishers allow applications to create DataWriters.
- Subscribers allow applications to create DataReaders
- DataWriters allow applications to publish (write) data.
- DataReaders allow applications to subscribe (read) data.

Figure 7.2 describes these entities DDS Entities and their relationship with the rest of objects involved in the DDS data-centric publish-subscribe model.

9.3.1.2 OPC UA Mapping Alternatives

There are different approaches to mapping the DDS Global Data Space to OPC UA. In general, we can categorize these in:

- Approaches mapping DDS Entities to OPC UA Objects with Methods and Variables similar to those specified by the DDS PIM. In other words, approaches that create an OPC UA PSM for DDS.
- Approaches mapping resources in the DDS Global Data Space such as Domains, Topics, and Instances to OPC UA Objects and Variables. These approaches rely on OPC UA Services to handle the operations that are usually performed by DDS Entities.

Each approach has advantages and disadvantages. On the one hand, mapping DDS Entities to OPC UA leverages the already existing DDS PIM that has been successfully ported to IDL, C++, and Java; but on the other hand, relying on custom Methods to perform operations equivalent to those provided by Services seems unnatural to OPC UA users and developers. Therefore, this specification has chosen the latter approach. It defines an OPC UA information model to represent the DDS Global Data Space, which simplifies interactions between OPC UA Clients and DDS applications by re-using the mechanisms that are most natural for them.

9.3.2 Representing DDS Domains in OPC UA

Figure 9.30 shows the Nodes and References involved in the mapping of DDS Domains to OPC UA.
9.3.2.1 Domain Objects

Domains shall be mapped to Object Nodes in the AddressSpace of the OPC UA server embedded in the OPC UA/DDS Gateway. Every Domain shall be modeled according to the DomainType ObjectType, which—as specified in sub clause 9.3.2.2—provides its basic structure and a reference to its DomainId. Moreover, Domain objects may contain references to a set of Topics representing the information DDS Publisher and Subscriber applications exchange in it.

Table 9.45 formally specifies the representation of a Domain in OPC UA using an Object Node.

Table 9.45: Domain Object Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>Domain_&lt;DomainId&gt;</td>
<td>BrowseName is composed of a Domain_ prefix and a numeric &lt;DomainId&gt;, representation of the 32-bit integer DomainId. For instance, the BrowseName of a Domain object representing Domain 0 shall be Domain_0.</td>
</tr>
<tr>
<td>IsAbstract</td>
<td>False</td>
<td>Objects representing Domains are never abstract.</td>
</tr>
<tr>
<td>References</td>
<td>NodeClass</td>
<td>BrowseName</td>
</tr>
<tr>
<td>HasTypeDefinition</td>
<td>ObjectType</td>
<td>DomainType</td>
</tr>
</tbody>
</table>

Figure 9.30: DDS Domain Mapping to OPC UA
### 9.3.2.2 DomainType ObjectType

To simplify the instantiation of new Domains, the OPC UA/DDS Gateway shall provide a DomainType ObjectType as specified in Table 9.46.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>DomainType</td>
<td>BrowseName of the DomainType ObjectType.</td>
</tr>
<tr>
<td>IsAbstract</td>
<td>False</td>
<td>DomainType objects are never abstract.</td>
</tr>
<tr>
<td>References</td>
<td>NodeClass</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BrowseName</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DataType</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TypeDefinition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ModelingRule</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.46: DomainType ObjectType Definition

Subtype of BaseObjectType.

<table>
<thead>
<tr>
<th>HasProperty</th>
<th>Variable</th>
<th>DomainId</th>
<th>Int32</th>
<th>PropertyType</th>
<th>Mandatory</th>
</tr>
</thead>
</table>

#### 9.3.3 Representing DDS Topics in OPC UA

Figure 9.31 shows the Nodes and References involved in the mapping of DDS Topics to OPC UA.

---

19 While the DDS specification states that the format of the DomainId is middleware-specific, the IDL PSM maps `DomainId_t` to a 32-bit integer.
9.3.3.1 Topic Objects

Topics shall be mapped to Object Nodes in the AddressSpace of the OPC UA Server embedded in the OPC UA/DDS Gateway. Every Topic shall be modeled according to the TopicType ObjectType, which provides its basic structure and a reference to its RegisteredTypeName.

Moreover, Nodes representing Topics shall provide references to Nodes representing their Instances. These are modeled using HasComponent references. Topics of keyed type may contain references to multiple Instance Nodes, whereas Topics of unkeyed types may contain a single reference to an Instance Node\(^{20}\).

Table 9.47 formally specifies the representation of a Topic in OPC UA using an Object Node.

Table 9.47: Topic Object Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>

\(^{20}\) As explained in sub clause 9.3.1.1, the data set associated with a Topic of unkeyed type is restricted to a single instance.
<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasTypeDefinition</td>
<td>ObjectType</td>
<td>TopicType</td>
<td>Every \textit{Topic} object shall be an instantiation of the \textit{TopicType} \textit{ObjectType}.</td>
</tr>
<tr>
<td>HasProperty</td>
<td>Variable</td>
<td>RegisteredTypeName</td>
<td>Every \textit{Topic} has an associated \textit{RegisteredType} identified by a \textit{RegisteredTypeName}. Upon instantiation, every \textit{Topic} object shall set the value of the \textit{RegisteredTypeName} property.</td>
</tr>
<tr>
<td>HasComponent*</td>
<td>Method</td>
<td>RegisterInstance</td>
<td>This method allows OPC UA Clients to register (i.e., create) new \textit{Instance Nodes} to represent DDS Instances. If the method is invoked successfully, a new \textit{Instance Node} is created and a \textit{HasComponent Reference} is added to the \textit{Topic Node} pointing to it. (*) This method is \textbf{only available in Topics with keyed types}. Topics with unkeyed types shall not have a \textit{RegisterInstance} method because there can only be a single Instance.</td>
</tr>
<tr>
<td>HasComponent*</td>
<td>Method</td>
<td>UnregisterInstance</td>
<td>This method allows OPC UA Clients to unregister Instances. (*) This method is \textbf{only available in Topics with keyed types}. Topics with unkeyed types shall not have an \textit{UnregisterInstance} method because there can only be a single Instance.</td>
</tr>
<tr>
<td>HasComponent*</td>
<td>Method</td>
<td>DisposeInstance</td>
<td>This method allows OPC UA Clients to dispose Instances. (*) This method is \textbf{only available in Topics with keyed types}. Topics with unkeyed types shall not have a \textit{DisposeInstance} method because there can only be a single Instance.</td>
</tr>
<tr>
<td>HasComponent</td>
<td>Variable</td>
<td>Instance[_&lt;InstanceId&gt;]</td>
<td>A \textit{Topic} may refer to one or more \textit{Variables} or \textit{Objects} representing instances of the top-level type (i.e., not nested type) it is associated with. \textit{Topics} of keyed types shall refer to \textit{Instance Nodes} representing instances that: (1) have been discovered by the \textit{DataReader} embedded in the Gateway, (2) have been registered via the \textit{RegisterInstance} method, or (3) have been instantiatiated via configuration files. \textit{Topics} of unkeyed types shall refer to a single \textit{Instance Node} representing their only instance. This \textit{Instance Node} shall be instantiated at startup time and shall always be available—even if no data has been received yet.</td>
</tr>
</tbody>
</table>

… …
9.3.3.1 RegisterInstance Method

RegisterInstance provides a mechanism to create new Instance Nodes. This Method shall only be provided by Topics with a keyed type.

The signature of RegisterInstance depends on the key members of the Topic type. It shall be set according to the following pattern:

```plaintext
StatusCode RegisterInstance {
    in <EquivalentType> <key_member_1_name>;
    [...in <EquivalentType> <key_member_N_name>);
}
```

Every key member of the type shall be mapped to an input parameter where:

- `<EquivalentType>`—The Type of the input parameter shall be the equivalent type according to the rules specified in clause 9.2.
- `<key_member_N_name>`—The Name of the input parameter shall be the fully-qualified name of the primitive member within the parent type. Nesting levels shall be represented by a double underscore: “__”. (e.g., for a structure key member `instance_identifier` containing a `type` string, the input parameter would be labeled as “`instance_identifier__type`”).

The Method shall return one of the following StatusCodes:

- **Good**—The operation was successful.
- **Bad_InvalidArgument**—One or more arguments are invalid.
- **Bad_NodeExists**—The Node to be created as a consequence of the invocation to RegisterInstance already exists.

Table 9.48 formally specifies the AddressSpace representation of the RegisterInstance Method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>RegisterInstance</td>
</tr>
<tr>
<td>References</td>
<td></td>
</tr>
<tr>
<td>NodeClass</td>
<td></td>
</tr>
<tr>
<td>BrowseName</td>
<td></td>
</tr>
<tr>
<td>DataType</td>
<td></td>
</tr>
<tr>
<td>TypeDefinition</td>
<td></td>
</tr>
<tr>
<td>ModelingRule</td>
<td></td>
</tr>
</tbody>
</table>

9.3.3.1.2 UnregisterInstance Method

UnregisterInstance provides a mechanism to unregister Instances. This Method shall only be provided by Topics with a keyed type.

The signature of UnregisterInstance depends on the key members of the Topic type. It shall be set according to the following pattern:

```plaintext
StatusCode UnregisterInstance {
    in <EquivalentType> <key_member_1_name>;
    [...in <EquivalentType> <key_member_N_name>];
}
```

Every key member of the type shall be mapped to an input parameter where:

For more information on the use cases that motivate the creation of this Method refer to sub clause 9.3.4.6.
• `<EquivalentType>`—The Type of the input parameter shall be the equivalent type according to the rules specified in clause 9.2.

• `<key_member_N_name>`—The Name of the input parameter shall be the fully-qualified name of the primitive member within the parent type. Nesting levels shall be represented by a double underscore: “__”. (e.g., for a structure key member `instance_identifier` containing a `type` string, the input parameter would be labeled as “`instance_identifier__type`”).

The Method shall return one of the following StatusCodes:

• **Good**—The operation was successful.

• **Bad_InvalidArgument**—One or more arguments are invalid.

Table 9.49 formally specifies the AddressSpace representation of the UnregisterInstance Method.

### Table 9.49: UnregisterInstance Method Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>UnregisterInstance</td>
</tr>
<tr>
<td>References</td>
<td></td>
</tr>
<tr>
<td>NodeClass</td>
<td></td>
</tr>
<tr>
<td>BrowseName</td>
<td></td>
</tr>
<tr>
<td>DataType</td>
<td></td>
</tr>
<tr>
<td>TypeDefinition</td>
<td></td>
</tr>
<tr>
<td>ModelingRule</td>
<td></td>
</tr>
<tr>
<td>HasProperty</td>
<td>Variable</td>
</tr>
<tr>
<td>InputArguments</td>
<td>Argument[]</td>
</tr>
<tr>
<td>PropertyType</td>
<td></td>
</tr>
<tr>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

9.3.3.1.3 DisposeInstance Method

`DisposeInstance` provides a mechanism to dispose Instances. This Method shall only be provided by Topics with a keyed type.

The signature of `DisposeInstance` depends on the key members of the Topic type. It shall be set according to the following pattern:

```plaintext
StatusCode DisposeInstance {
    in <EquivalentType> <key_member_1_name>; 
    [...in <EquivalentType> <key_member_N_name>];
}
```

Every key member of the type shall be mapped to an input parameter where:

• `<EquivalentType>`—The Type of the input parameter shall be the equivalent type according to the rules specified in clause 9.2.

• `<key_member_N_name>`—The Name of the input parameter shall be the fully-qualified name of the primitive member within the parent type. Nesting levels shall be represented by a double underscore: “__”. (e.g., for a structure key member `instance_identifier` containing a `type` string, the input parameter would be labeled as “`instance_identifier__type`”).

The Method shall return one of the following StatusCodes:

• **Good**—The operation was successful.

• **Bad_InvalidArgument**—One or more arguments are invalid.

Table 9.50 formally specifies the AddressSpace representation of the DisposeInstance Method.

### Table 9.50: DisposeInstance Method Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To simplify the instantiation of new Topics, the OPC UA/DDS Gateway shall provide a TopicType ObjectType as specified in Table 9.51.

### Table 9.51: TopicType ObjectType Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>TopicType</td>
<td>BrowseName of the TopicType ObjectType.</td>
</tr>
<tr>
<td>IsAbstract</td>
<td>False</td>
<td>TopicType objects are never abstract.</td>
</tr>
<tr>
<td>Reference</td>
<td>NodeClass</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BrowseName</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DataType</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TypeDefinition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ModelingRule</td>
<td></td>
</tr>
</tbody>
</table>

Subtype of BaseObjectType.

<table>
<thead>
<tr>
<th>HasProperty</th>
<th>Variable</th>
<th>RegisteredTypeName</th>
<th>String</th>
<th>PropertyType</th>
<th>Mandatory</th>
</tr>
</thead>
</table>

9.3.4 Representing DDS Instances and Samples in OPC UA

#### 9.3.4.1 DDS Instance Node Representation

DDS Topic Instances shall be mapped to OPC UA Variable or Object nodes representing instances of the associated type in the Gateway according to the rules specified in clause 9.2.

Figure 9.32 shows the Nodes and References involved in the definition of an Instance, excluding those introduced by the aforementioned mapping rules.
The `BrowseName` of the `Variable` or `Object Node` is different for `Instances` of `Topics` of keyed and unkeyed types:

- The `BrowseName` of the single `Instance Node` of a `Topic` of unkeyed type shall be “`Instance`”.
- The `BrowseName` of the `Instance Nodes` of a `Topic` of keyed type, shall be constructed according to the following convention: “`Instance_<InstanceId>`”. Where `<InstanceId>` is an undefined string representing the value of `DDS::InstanceHandle_t` returned by the `DataReader`’s `get_key_value()` operation (see sub clause 2.2.2.5.3.29 of [DDS]).

Because at the time of writing of this document the format for `DDS::InstanceHandle_t` is undefined in the [DDS] specification, we may only propose a number of non-normative string representations alternatives\(^\text{22}\).

For instance, if all key fields of the type are of numeric or string types, `<InstanceId>` may be a combination of the string representation of the value of all key fields separated by colons (`:`). Alternatively, `<InstanceId>` may be the MD5 hash of the value of a vendor’s implementation of `DDS::InstanceHandle_t`.

Besides the `References` defined by the mapping rules specified in clause 9.2 for the type, the OPC UA `Variable` or `Object Nodes` representing `Instances` shall also include a number of `HasProperty References` to `Variables` of `PropertyType` representing a subset of the fields of the `DDS::SampleInfo` structure\(^\text{23}\). These fields provide important metadata information about the state of the instance and the samples that have been received by `DataReaders` embedded in the Gateway.

Table 9.52 provides the list of `Variables` of `PropertyType` that every `Instance Node` shall refer to. Note that all these `Variables` shall be marked as read-only; i.e., they shall be instantiated with the `WriteMask Attribute` set to 0.

---
\(^{22}\) This is consistent with the approach taken by other specifications such as Web-Enabled DDS, which defines the value of `DDS::InstanceHandle_t` as an opaque string that can be used to refer to a registered instance.

\(^{23}\) In particular, this specification has chosen the same subset of fields specified in Web-Enabled DDS [DDS-WEB].
Table 9.52: PropertyType Variables Representing Members of DDS::SampleInfo

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>DataType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleState</td>
<td>String</td>
<td>String representation of the state of a sample. Implementers of this specification shall assign SampleStates to strings as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• READ: “READ”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NOT_READ: “NOT_READ”</td>
</tr>
<tr>
<td>ViewState</td>
<td>String</td>
<td>String representation of the ViewState of a sample. Implementers of this specifications shall assign ViewStates to strings as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NEW: “NEW”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NOT_NEW: “NOT_NEW”</td>
</tr>
<tr>
<td>InstanceState</td>
<td>String</td>
<td>String representation of the state of a given instance. Implementers of this specifications shall assign InstanceStates to strings as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ALIVE: “Alive”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NOT_ALIVE_DISPOSED: “NOT_ALIVE_DISPOSED”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NOT_ALIVE_NO_WRITERS: “NOT_ALIVE_NO_WRITERS”</td>
</tr>
<tr>
<td>SourceTimestamp</td>
<td>DateTime</td>
<td>Date-Time representation of the source timestamp for a given sample. Implementers of this specification shall handle the conversion from DDS::Time_t to OPC UA’s Date-Time.</td>
</tr>
<tr>
<td>InstanceHandle</td>
<td>String</td>
<td>String representation of the DDS::InstanceHandle_t according to the rules specified in sub clause 9.3.4.1 of this specification.</td>
</tr>
<tr>
<td>ValidData</td>
<td>Boolean</td>
<td>Boolean value indicating whether there is data associated with a given sample.</td>
</tr>
</tbody>
</table>

Table 9.53 formally specifies the definition of an Instance Node according to the rules mentioned above.

Table 9.53: Instance Variable or Object Node Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrowseName</td>
<td>&lt;String&gt;</td>
<td>String with the name of the Instance the Node represents. This string shall be constructed as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For Nodes representing the single instance of a Topic with an unkeyed type, BrowseName shall be “Instance”.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For Nodes representing an Instance of a Topic with a keyed type, BrowseName shall be “Instance_&lt;InstanceId&gt;”. Where</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;InstanceId&gt; is an undefined string identifying the instance. For example, &lt;InstanceId&gt; may be the string representation or the</td>
</tr>
</tbody>
</table>

24 To simplify the mapping of SampleState, ViewState, and InstanceState we have chosen a string representation rather than an enumeration, which requires the definition of a new type and adds an extra level of indirection for client applications.
Attributes specific to the NodeClass (Variable or Object) of the Instance Node. These attributes shall be configured as specified in the mapping rules defined in clause 9.2 for instances of the DDS type this Instance Node represents.

<table>
<thead>
<tr>
<th>References</th>
<th>NodeClass</th>
<th>BrowseName</th>
<th>DataType</th>
<th>TypeDefinition</th>
<th>ModelingRule</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasProperty</td>
<td>Variable</td>
<td>SampleState</td>
<td>String</td>
<td>PropertyType</td>
<td>Mandatory</td>
</tr>
<tr>
<td>HasProperty</td>
<td>Variable</td>
<td>ViewState</td>
<td>String</td>
<td>PropertyType</td>
<td>Mandatory</td>
</tr>
<tr>
<td>HasProperty</td>
<td>Variable</td>
<td>InstanceState</td>
<td>String</td>
<td>PropertyType</td>
<td>Mandatory</td>
</tr>
<tr>
<td>HasProperty</td>
<td>Variable</td>
<td>SourceTimestamp</td>
<td>DateTime</td>
<td>PropertyType</td>
<td>Mandatory</td>
</tr>
<tr>
<td>HasProperty</td>
<td>Variable</td>
<td>InstanceHandle</td>
<td>String</td>
<td>PropertyType</td>
<td>Mandatory</td>
</tr>
<tr>
<td>HasProperty</td>
<td>Variable</td>
<td>ValidData</td>
<td>Boolean</td>
<td>PropertyType</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

List of references derived from the mapping rules specified in clause 9.2 for instances of the DDS type the Instance Node represents.
For example, these may include:

- A HasTypeDefinition Reference to BaseDataVariableType, BaseObjectType, or any other VariableType or ObjectType Node.
- HasComponent References, such as those that link a Structure with Nodes representing its members.
- HasOrderedComponent References, such as those that link a Map Object to its MapEntries.

9.3.4.2 Updating the Value of DDS Instance Nodes

The OPC UA/DDS Gateway shall update the Value of the Variable Nodes associated with every instance of every Topic—including the Property Variables representing the DDS::SampleInfo structure—with the content of the latest samples received by its internal DataReaders. As already mentioned, there are some distinctions regarding Topics of keyed and unkeyed types:

- For Topics of unkeyed types, the only Instance Node shall be updated with the latest sample available for that Topic.
- For Topics of keyed types, the different Instance Nodes shall be updated with the latest sample available for that specific Topic instance.

The Value Variable Nodes associated with an Instance shall be updated as follows:

- The Value of the Variable Node (or Variable Nodes) representing sample data (i.e., all Variable Nodes except the Property Variables listed in Table 9.52) shall only be updated if the valid_data flag of the DDS::SampleInfo structure is true.
• The Value of the Variable Nodes representing sample info (i.e., the Property Variables listed in Table 9.52) shall be updated regardless of the value of the valid_data flag\textsuperscript{25}.

Implementations of this specification shall provide users with the necessary means to configure the QoS Policies associated with the internal DataReaders. This specification provides an optional conformance point with a configuration syntax for this purpose in chapter 10.

Optionally, implementers may provide additional mechanisms to automatically remove Instance Nodes representing NOT_ALIVE instances (i.e., Instances whose InstanceState is NOT_ALIVE_DISPOSED or NOT_ALIVE_NO_WRITERS).

9.3.4.3 Reading and Monitoring Instance Nodes

OPC UA Clients may use the Read Service to read the current value of Instances of a DDS Topic by invoking the appropriate operation on the Variable Nodes representing the Value associated with an Instance Node.

Moreover, OPC UA Clients may use Services of the Subscription and MonitoredItems Service Sets to receive updates any time the value of one of the Variable Nodes representing the Value associated with an Instance Node changes.

9.3.4.4 Reading Historical Data from Instance Nodes

OPC UA Clients may use the HistoryRead Service to read historical values on a specific DDS Topic Instance.

To enable that scenario, the OPC UA Server embedded in the Gateway shall instantiate the Variable Nodes associated with every Instance of the DDS Topic as HistoricalDataNodes. As specified in sub clause 5.2.5 of [OPCUA-11]), this implies defining—setting to 1—both the Historizing Attribute and the HistoryRead bit in the AccessLevel Attribute of every Variable Node. These Attributes—along with the OPC UA Server’s HistoryServerCapabilities object—inform Client applications of the availability of historical access. Additionally, the Server may add a HasHistoricalConfiguration Reference to a “HA Configuration” Node indicating the desired HistoricalConfiguration for every Variable. The selected “HA Configuration” shall be consistent for all Variable Nodes associated with every Instance of the DDS Topic.

Moreover, the DataReader embedded in the Gateway to handle subscription to the DDS Topic shall be configured to support historical access. In particular, their HISTORY QoS Policy shall be configured either as KEEP_ALL or as KEEP_LAST with a HISTORY_DEPTH big enough to store the desired time span of samples. Implementers of this specification shall provide users with the means to configure these QoS Policies (see chapter 10).

9.3.4.5 Writing Instance Nodes

OPC UA Clients may use the Write Service to update the value of any of the Variable Nodes associated with an Instance Node. This sub clause describes the behavior of the OPC UA/DDS Gateway to facilitate those updates.

Updates on the Value of Variables associated with Instance Nodes shall be trigger the invocation of the write() method on a DDS DataWriter instantiated by the OPC UA/DDS Gateway for that purpose. It is up to implementers of the specification to decide whether to invoke the write() operation immediately or wait until a certain number of updates have been received. This allows optimizations such batching of updates to members of a specific structure before calling write().

Updates on Variables representing key members of the data type associated with a Topic are disallowed because they would automatically transform the existing Instance into a different Topic Instance. In the case of key union

\textsuperscript{25} This enables OPC UA client applications to receive updated information about the lifecycle of an instance. For example, it provides information on whether the instance is ALIVE or NOT_ALIVE (DISPOSED or NO_WRITERS).
discriminators this is not a problem, because their value is not exposed in the AddressSpace of the OPC UA/DDS. However, key structure members shall be explicitly configured as read-only. As specified in 9.2.4.1.2, to allow generic and non-generic OPC UA Clients to access the value of the different members of structure, these are represented twice in the AddressSpace of the OPC UA Server. Therefore their immutability must be specified and enforced differently:

1. For updates on key members of a Variable of Structure DataType, the Gateway shall validate that the Write operation does not change their value and return StatusCode Bad_UserAccessDenied otherwise.

2. For updates on Variable Nodes representing members of a structure linked to Variable Nodes representing the structure with a HasComponent Reference, the Gateway shall rely on the behavior of the underlying OPC UA SDK by definition—the WriteMask Attribute is set to zero (read-only) as specified by the mapping rules in sub clause 9.2.8.

Likewise, updates on the PropertyType Variables representing members of the DDS::SampleInfo structure (i.e., SampleState, ViewState, InstanceState, SourceTimestamp, InstanceHandle, and ValidData) are disallowed because, as stated in sub clause 9.3.4.1, the WriteMask Attribute of these nodes shall be set to zero.

Finally, the Value of Variables associated with an Instance Node shall be updated in the AddressSpace after the corresponding DDS DataWriter has called the write() operation. The OPC UA/DDS Gateway shall ensure that the value of structure members—which are represented twice in the AddressSpace of the OPC UA Server—remains consistent.

9.3.4.6 Registering New Instances

Occasionally, OPC UA Clients may wish to use the Writer Service to write a new sample of an Instance that has not previously been registered. In other words, they may wish to update the value of an Instance Node that does not exist in the AddressSpace of the OPC UA Server.

To register an instance, OPC UA Clients must invoke the RegisterInstance Method associated with the corresponding Topic Object using the Call Service [OPCUA-04]. This Method—defined in sub clause 9.3.3.1.1—is only available in Topics with keyed types. (Topics of unkeyed types always have an Instance Node associated with it that can be used to write any sample of that Topic.) The InputParameters for the RegisterInstance Method are the fields that represent the key; therefore, the OPC UA Client shall pass in the appropriate values for the Instance to be registered.

After invoking the Method, the Client application will receive a StatusCode indicating the success or failure of the operation. If StatusCode is Good, then the OPC UA/DDS Gateway will create a new Instance Node representing the registered instance in the AddressSpace of its OPC UA Server, and will link it to the Topic Node with a HasComponent Reference. Client applications may now use the Write Service to write samples of the new instance, or the Read Service to read the most recent value of the Instance.

9.3.4.7 Unregistering and Disposing Instances

OPC UA Clients that may wish to unregister or dispose an Instance can use the corresponding Method associated with the Topic. Like in the case of RegisterInstance, these Methods are only available in Topics with keyed types.

9.3.5 Implementation Considerations

9.3.5.1 OPC UA Implementation Considerations

The representation of the DDS Global Data Space specified in this chapter requires the OPC UA/DDS Gateway to embed an OPC UA Server. This OPC UA Server shall be capable of:
• Instantiating a number of Nodes in its AddressSpace to represent DDS types, Domains, Topics, and Instances that OPC UA Client applications may browse, read, and write to participate as a first-class citizen in the DDS Global Data Space.

• Responding to View Service requests from OPC UA Clients willing to browse the AddressSpace of the Server.

• Responding to Read Service requests from OPC UA Clients willing to read the current value of a mapped DDS Topic Instance (see sub clause 9.3.4.3).

• Responding to HistoryRead Service requests from OPC UA Clients willing to read historical values of a mapped DDS Topic Instance (see sub clause 9.3.4.4).

• Responding to Write Service requests from OPC UA Clients willing to publish data on a mapped DDS Topic (see sub clause 9.3.4.5).

• Responding to Subscription and MonitoredItems Service requests from OPC UA Clients willing to subscribe to the mapped DDS Topics to receive updates on data changes (see sub clause 9.3.4.3).

• Being discovered by the Local and Global Discovery Servers defined in [OPCUA-12].

To comply with all the requirements listed above, the OPC UA Server shall comply with the Embedded UA Server Profile defined in sub clause 6.5.54 of [OPCUA-07]. Additionally, to support access to historical data, the OPC UA Server shall comply with the Historical Raw Data Server Facet defined in sub clause 6.5.36 of [OPCUA-07]. Consequently, compliant implementations of this specification shall be built on top of an OPC UA implementation capable of passing the conformance tests specified for those profiles and facets by the OPC Foundation.

Lastly, it is important to note that implementers of this specification may need to configure the underlying OPC UA Server to require authentication, access control, and encryption using the mechanisms provided by the OPC UA Security Model specified in [OPCUA-02]. These mechanisms can be used to enforce that only authorized OPC UA Clients can access the AddressSpace of the OPC UA Server, and therefore the DDS Global Data Space—or a subset of it. These mechanisms may pose additional requirements in the underlying OPC UA Servers, which shall be addressed according to the needs of each specific use case.

9.3.5.2 DDS Implementation Considerations

The OPC UA/DDS Gateway shall be capable of publishing and subscribing to updates in the DDS Global Data Space using a DDS implementation complaint with:

- Minimum Profile of [DDS]
- Statements listed in clause 8.4.2 of [DDSI-RTPS].

Some deployments may require using the mechanisms specified in [DDS-SECURITY] to access information provided by secured DDS applications, or publish information in restricted Domains. In those cases, the underlying DDS implementation shall also be compliant with the Built-in Plugin Interoperability and Plugin Framework Conformance Points of [DDS-SECURITY].

As specified in the rest of clauses dealing with DDS and OPC UA integration, the Gateway shall be capable of dealing with two different security models: the OPC UA Security Model on one end and the DDS Security Model on the other end. Each security model shall be configured separately depending on the needs of the end user of the OPC UA/DDS Gateway.
10 OPC UA/DDS Gateway Configuration

This chapter defines an XML syntax to configure the OPC UA/DDS Gateway. It is built upon the DDS Consolidated XML Syntax [DDS-XML], which provides all the necessary constructs to specify DDS resources in XML.

10.1 Overview

The syntax to configure the OPC UA/DDS Gateway is specified in two normative XSD files.

- `dds-opcua_definitions_nonamespace.xsd`—Contains all the type definitions that build up the XML syntax to configure the Gateway. It makes use of `dds-xml_domainparticipant_definitions_nonamespace.xsd`, a schema file specified in the DomainParticipants Building Block of [DDS-XML] that provides syntax to represent DDS types, entities, and QoS Policies. Moreover, to facilitate the integration of the definitions into more complex or vendor-specific schema files, the XSD file defines neither a root element nor namespaces.

- `dds-opcua_configuration.xsd`—Defines the root element of the OPC UA/DDS Gateway configuration file and the `http://www.omg.org/spec/DDS-OPCUA` namespace. It includes `dds-opcua_definitions_nonamespace.xsd` to resolve the necessary type definitions. This is the schema file that shall be used to validate OPC UA/DDS Gateway XML configuration files.

10.2 Configuration

Table 10.1 provides implementers of this specification with an overview of the configuration elements that are part of the OPC UA/DDS Gateway XML configuration syntax. All described elements—except the noted exceptions—are defined in `dds-opcua_definitions_nonamespace.xsd`. Attributes and low-level configuration details have been left out of this overview; therefore, implementers shall refer to the normative XSD file for a comprehensive study of all the configuration capabilities of the syntax defined by this specification.

<table>
<thead>
<tr>
<th>XML Configuration Element</th>
<th>Type Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dds&gt;</code></td>
<td><code>rootType</code></td>
<td>Root element. Is the entry point of the OPC UA/DDS Gateway configuration.</td>
</tr>
<tr>
<td><code>&lt;types&gt;</code></td>
<td><code>types</code></td>
<td>Defines types that <code>DomainParticipants</code> may register to create <code>Topics</code> for reading or writing DDS data.</td>
</tr>
<tr>
<td><code>&lt;qos_libraries&gt;</code></td>
<td><code>qosLibrary</code></td>
<td>Organizes QoS Profiles with QoS Policies that may be used to specify behavior of the DDS entities instantiated by the Gateway.</td>
</tr>
<tr>
<td><code>&lt;ddsopcua_gateway&gt;</code></td>
<td><code>ddsOpcUaGateway</code></td>
<td>Configures of an OPC UA/DDS Gateway that may be instantiated by the application or library implementing it. A <code>ddsopcua_gateway</code> configuration may refer to <code>types</code> and <code>qos_libraries</code> specified in the configuration file. Moreover, it may define...</td>
</tr>
</tbody>
</table>

---

26 This allows applying the Chameleon Schema pattern defined in [DDS-XML].

27 `types` is defined in the schema file associated with Types Building Block of [DDS-XML].

28 `qosLibrary` is defined in the schema file associated with the QoS Building Block of [DDS-XML].
<table>
<thead>
<tr>
<th>XML Configuration Element</th>
<th>Type Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcua_connections, opcua_servers, domain_participants, opcua_to_dds_bridges; and dds_to_opcua_bridges</td>
<td></td>
<td>The definition of multiple bridges—on either direction—in the same instance of the OPC UA/DDS Gateway is permitted.</td>
</tr>
<tr>
<td>&lt;opcua_connection&gt;</td>
<td>opcuaConnection</td>
<td>Defines a connection of the OPC UA/DDS Gateway to an external service. When referenced from a service_set or subscription configuration in the context of an OPC UA to DDS Bridge, the Gateway will instantiate an OPC UA Client capable of connecting to the specified Server according to the specified configuration. An OPC UA/DDS Gateway configuration may contain multiple opcua_connection definitions.</td>
</tr>
<tr>
<td>&lt;opcua_server&gt;</td>
<td>opcuaServer</td>
<td>Defines an OPC UA Server that may be instantiated by DDS to OPC UA Bridges. The AddressSpace of these servers will expose the DDS Global Data Space to OPC UA Clients. The configuration of OPC UA Servers is unspecified as those settings are not standardized and are therefore OPC UA vendor-specific. An OPC UA/DDS Gateway configuration may contain multiple opcua_server definitions.</td>
</tr>
<tr>
<td>&lt;domain_participant&gt;</td>
<td>ddsDomainParticipant</td>
<td>Configures a DomainParticipant, which provides the entry point for OPC UA to DDS or DDS to OPC UA Bridges to operate in a DDS Domain. The same DomainParticipant definition may be used by different bridges regardless of their direction. An OPC UA/DDS Gateway configuration may contain multiple domain_participant definitions.</td>
</tr>
<tr>
<td>&lt;opcua_to_dds_bridge&gt;</td>
<td>opcua2DdsBridge</td>
<td>Configures an OPC UA to DDS Bridge, which exposes the AddressSpace of one or more OPC UA Servers to DDS applications. An OPC UA/DDS Gateway configuration may contain multiple opcua_to_dds_bridge definitions.</td>
</tr>
<tr>
<td>&lt;service_set&gt;</td>
<td>opcuaServiceSet</td>
<td>Exposes selected OPC UA Services from an OPC UA Server to DDS applications by creating equivalent DDS Services using RPC over DDS, as specified in clause 8.3. An OPC UA to DDS Bridge may include multiple service_set definitions to expose Service Sets from different OPC UA Servers to DDS</td>
</tr>
<tr>
<td>XML Configuration Element</td>
<td>Type Definition</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>&lt;subscription&gt;</td>
<td>opcuaSubscription</td>
<td>Defines OPC UA Inputs (Subscriptions to different MonitoredItems—DataItems and EventItems—in OPC UA Servers) and DDS Outputs (DataWriters associated to DDS Topics) and provides the ability to map DataItems or EventItems from different OPC UA Inputs to fields of Topics associated with DDS Outputs. An OPC UA to DDS Bridge may include multiple subscription definitions.</td>
</tr>
<tr>
<td>&lt;opcua_input&gt;</td>
<td>opcuaInput</td>
<td>Configures a Subscription to an OPC UA Client and a set of MonitoredItems—DataItems or EventItems—using an opcua_connection definition. A subscription may contain different opcua_input definitions to allow combining information from different Inputs in one or more DDS Outputs.</td>
</tr>
<tr>
<td>&lt;dds_output&gt;</td>
<td>ddsOutput</td>
<td>Configures a DDS DataWriter capable of publishing a Topic in the context of an already defined domain_participant. The definition of a dds_output does not trigger any publication; for that to happen, users shall specify mappings and assignments of elements in an OPC UA Input to fields of the Topic associated with an OPC UA Output. A subscription may contain different dds_output definitions.</td>
</tr>
<tr>
<td>&lt;mapping&gt;</td>
<td>inputOutputMapping</td>
<td>Maps DataItems and EventItems from an OPC UA Input to fields of one or more DDS Outputs. A subscription shall contain a single mapping definition. In other words, only one mapping section can appear under a subscription element.</td>
</tr>
<tr>
<td>&lt;assignment&gt;</td>
<td>inputOutputAssignment</td>
<td>Assigns DataItems, EventFields from an EventItem, or a constant values to fields of the Topic associated with a DDS Output. Each assignment is therefore bound to a specific DDS Output. A reference to an OPC UA Input under the subscription is also required. The referred OPC UA Input is used as the default input for all the MonitoredItems being assigned (DataItems or EventFields); however, in the mapping of specific fields, users are allowed to override the default OPC UA Input by referencing a different Input from the subscription. This enables combining information from different OPC UA Inputs into a</td>
</tr>
<tr>
<td>XML Configuration Element</td>
<td>Type Definition</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>single OPC UA Output.</td>
<td></td>
<td>A <strong>mapping</strong> definition may contain multiple assignments—as many as DDS Outputs under the parent <strong>subscription</strong> definition.</td>
</tr>
<tr>
<td>&lt;dds_to_opcua_bridge&gt;</td>
<td>dds2opcuaBridge</td>
<td>Configures a DDS to OPC UA Bridge, which instantiates an OPC UA <strong>Server</strong> capable of representing the DDS Global Data Space in its <strong>AddressSpace</strong>. On one side, the DDS to OPC UA Bridge must refer to one of the <strong>opcua_server</strong> definitions of the configuration file; on the other side, the Bridge must refer to one or multiple <strong>domain_participant</strong> definitions (which provide access to one or multiple DDS Domains). An OPC UA/DDS Gateway configuration may contain multiple <strong>dds_to_opcua_bridge</strong> definitions.</td>
</tr>
<tr>
<td>&lt;domain&gt;</td>
<td>ddsDomain</td>
<td>Configures a <strong>Domain</strong> that shall be added to the OPC UA <strong>Server</strong> associated with the parent <strong>dds_to_opcua_bridge</strong> definition. The configuration shall reference a <strong>domain_participant</strong> to access the <strong>Domain</strong>. A <strong>dds_to_opcua_bridge</strong> may contain several <strong>domain</strong> definitions to represent different <strong>Domains</strong>.</td>
</tr>
<tr>
<td>&lt;topic&gt;</td>
<td>ddsTopic</td>
<td>Configures a <strong>Topic</strong> to be exposed in the <strong>AddressSpace</strong> of the OPC UA <strong>Server</strong> embedded into the Gateway. A <strong>domain</strong> may contain several <strong>topic</strong> definitions to represent different <strong>Topics</strong> available in the <strong>Domain</strong>.</td>
</tr>
<tr>
<td>&lt;registered_type_name&gt;</td>
<td>xs:string</td>
<td>Name of the type—previously registered with the <strong>DomainParticipant</strong>—the <strong>Topic</strong> will be associated with.</td>
</tr>
<tr>
<td>&lt;read_access&gt;</td>
<td>ddsReadAccess</td>
<td>Provides mechanisms to: (1) enable read access on the OPC UA <strong>Nodes</strong> associated with the <strong>Topic</strong>, (2) configure the associated <strong>DataReader</strong>, and (3) define content filters that can be used, among other things, to specify which <strong>Topic</strong> Instances are exposed to OPC UA <strong>Clients</strong>.</td>
</tr>
<tr>
<td>&lt;write_access&gt;</td>
<td>ddsWriteAccess</td>
<td>Provides mechanism to: (1) enable write access on the OPC UA <strong>Nodes</strong> associated with the <strong>Topic</strong>, (2) per-register <strong>Topic</strong> Instances that <strong>Clients</strong> may write, and (3) configure the associated <strong>DataWriter</strong>’s QoS.</td>
</tr>
<tr>
<td>&lt;topic_group&gt;</td>
<td>ddsTopicGroup</td>
<td>Configures a group of <strong>Topics</strong> to be exposed in the <strong>AddressSpace</strong> of the OPC UA <strong>Server</strong> embedded into</td>
</tr>
</tbody>
</table>
the Gateway. In particular, they provide the ability to expose topics matching a certain criteria in terms of topic name and topic type.

A domain may contain several topic_group definitions to represent different topics available in the Domain.

A regular expression describing which topics should be represented in the AddressSpace of the server. Topics with names that matching this filter are allowed to be represented, unless they do not pass the additional filters.

A regular expression describing which topics should be represented in the AddressSpace of the OPC UA Server. This is applied after the allow filter.

A regular expression describing a set of type names registered in the DDS DomainParticipant. Topics with data types that match this filter are allowed to be shown in the AddressSpace of the OPC UA Server.

A regular expression describing a set of type names registered in the DDS DomainParticipant that shall be filtered out. Topics with data types that match this regular expression are not allowed to be shown in the AddressSpace of the OPC UA Server. This is applied after allow_type_name_filter.

See definition above.

See definition above.

### 10.3 Examples (non-normative)

This specification includes two non-normative XML files that illustrate different configurations of the OPC UA/DDS Gateway according to the syntax specified in this chapter.

#### 10.3.1 OPC UA to DDS Bridge Example

This example illustrates how to configure the OPC UA/DDS Gateway to leverage the mappings specified in clauses 8.3 and 8.4. Effectively, it builds a bridge between the AddressSpace of an OPC UA Server and DDS applications.

At a high level, the XML configuration document is organized as follows:

```xml
<dds>
```

OPC UA/DDS Gateway 1.0
<types>
  <struct>...</struct>
</types>

<ddsopcua_gateway name="MyGateway">
  <opcua_connection>...</opcua_connection>
  <domain_participant>...</domain_participant>
  <opcua_to_dds_bridge>
    <service_set>...</service_set>
    <subscription>
      <opcua_input>...</opcua_input>
      <dds_output>...</dds_output>
      <dds_output>...</dds_output>
      <dds_output>...</dds_output>
      <mapping>...</mapping>
    </subscription>
  </opcua_to_dds_bridge>
</ddsopcua_gateway>
</dds>

Where:

- `<types>` defines DDS types that are required to create DDS Outputs according to the users’ interests and the mapping rules defined in sub clause 8.4.2.
- `<ddsopcua_gateway>` defines a scenario to be loaded by the Gateway. Each definition includes connections to OPC UA Servers and DDS DomainParticipants that may be used to create DDS Topics.
- `<opcua_to_dds_bridge>` configures OPC UA Service Set and Subscription mappings to build a bridge between the AddressSpace of OPC UA Servers and DDS applications.

The complete example may be found in the non-normative file `dds-opcua_opcua2dds_configuration.xml`, which is included with this specification.

### 10.3.1.1 DDS Type Definitions

Following the mapping rules specified in sub clause 8.4.2, we define the DDS types that we will use in each DDS Output. In particular, we have decided to create three data types to group the set of MonitoredItems in a meaningful set of Topics: MotorStatus, DevicePosition, and Event. The DDS types associated with those Topics are represented in XML format using the syntax specified in [DDS-XML].

The **MotorDataType** is defined as follows:

```xml
<struct name="MotorDataType">
  <member name="motor_name" type="string" key="true" />
  <member name="motor_moves" type="boolean" />
  <member name="motor_changes_direction" type="boolean" />
</struct>
```

Note that it includes an extra member named `motor_name` that identifies the source of information and serves as a key.

The **DevicePositionType** is defined as follows:

```xml
<struct name="DevicePositionType">
  <member name="device_name" type="string" key="true" />
  <member name="longitude" type="float64" />
  <member name="latitude" type="float64" />
  <member name="altitude" type="float64" />
</struct>
```

Lastly, the **EventType** is defined as follows:

```xml
<struct name="EventType">
  <member name="message" type="string" />
</struct>
```
10.3.1.2 OPC UA Connection and DDS DomainParticipant Definition

To connect the OPC UA Gateway with OPC UA Servers and DDS Domains, we must first define an OPC UA Connection and a DDS DomainParticipant.

The OPC UA Connection is defined as follows:

```
<opcua_connection name="MyServerConnection" server_endpoint_url="opc.tcp://10.10.100.130:55001">
  <timeout>5000</timeout>
</opcua_connection>
```

When defining an OPC UA Connection we must provide the EndpointUrl of the remote Server we aim to connect to.

The DDS DomainParticipant is defined as follows:

```
<domain_participant name="MyDomainParticipant" domain_id="0">
  <register_type name="MotorDataType" type_ref="MotorDataType" />
  <register_type name="DeviceDataType" type_ref="DevicePosition" />
  <register_type name="EventType" type_ref="EventType" />
</domain_participant>
```

When defining the DomainParticipant, we must register all the types we are going to use in the deployment. In this case, we register those that describe the MonitoredItems we want to send over DDS. Note that the DomainParticipantQos can be defined as a nested structure of the DomainParticipant.

10.3.1.3 OPC UA to DDS Bridge Definition

The OPC UA to DDS Bridge configures Service Sets and Subscriptions using one or more OPC UA Connections. In our example, we configure one Service Set mapping and one Subscription Mapping as follows.

```
<opcua_to_dds_bridge name="MyOpcUa2DdsBridge">
  <service_set>...</service_set>
  <subscription>...</subscription>
</opcua_to_dds_bridge>
```

It is important to note that multiple OPC UA to DDS Bridges (possibly along with multiple DDS to OPC UA Bridges) may be instantiated by a single OPC UA/DDS Gateway configuration.

10.3.1.4 OPC UA Service Set Mapping Definition

In our example, we map a subset of an OPC UA Server’s Services to an equivalent DDS Service as follows:

```
<service_set opcua_connection_ref="MyServerConnection" domain_participant_ref="MyDomainParticipant" >
  <view_service_set>
    <enabled>true</enabled>
  </view_service_set>
  <query_service_set>
    <enabled>false</enabled>
  </query_service_set>
  <attribute_service_set>
    <enabled>true</enabled>
  </attribute_service_set>
  <method_service_set>
    <enabled>false</enabled>
  </method_service_set>
</service_set>
```
On one side, we specify the OPC UA Connection to be used, which effectively indicates the OPC UA Server that is going to be exposed; and on the other side, the DomainParticipant under which all DDS entities will be created. We must explicitly enable every Service we want to expose.

Note that multiple Service Set Mapping definitions may be created under a single OPC UA to DDS Bridge.

### 10.3.1.5 OPC UA Subscription Mapping Definition

An OPC UA Subscription mapping defines OPC UA Inputs (subscriptions), DDS Outputs (publications), and Input/Output mappings (assignments).

```xml
<subscription name="MySubscription">
  <opcua_input name="MyInput"
    opcua_connection_ref="MyServerConnection">
    ...
  </opcua_input>
  ...
  <dds_output name="MotorDataPublication"
    domain_participant_ref="MyDomainParticipant">
    ...
  </dds_output>
  ...
  <dds_output name="DevicePublication"
    domain_participant_ref="MyDomainParticipant">
    ...
  </dds_output>
  ...
  <dds_output name="EventPublication"
    domain_participant_ref="MyDomainParticipant">
    ...
  </dds_output>
</subscription>
```

#### 10.3.1.5.1 OPC UA Input

The OPC UA Input in the example configures an OPC UA Subscription with a set of MonitoredItems and some properties associated with the SubscriptionProtocol. To create an OPC UA Input it is necessary to specify an OPC UA Connection.

##### 10.3.1.5.1.1 OPC UA Input and Subscription Protocol Definition

At a high level, the OPC UA Input and SubscriptionProtocol are defined as follows; below we provide a detailed description of each MonitoredItems associated with it:

```xml
<opcua_input name="MyInput"
  opcua_connection_ref="MyServerConnection">
  <subscription_protocol>
    <requested_publishing_interval>10</requested_publishing_interval>
    <requested_lifetime_count>3000</requested_lifetime_count>
    <requested_max_keep_alive_count>1000</requested_max_keep_alive_count>
    <max_notifications_per_publish>0</max_notifications_per_publish>
    <publishing_enabled>true</publishing_enabled>
    <priority>0</priority>
  </subscription_protocol>
  ...
  <monitored_items>
  
  </monitored_items>
</opcua_input>
```

#### 10.3.1.5.1.2 MonitoredItems

This section defines each of the MonitoredItems that are going to be attached to the OPC UA Input upon instantiation:
- **MotorMoves**—Boolean value indicating whether the motor is currently moving. In this case, the application is monitoring data changes on the *Value Attribute* of a *Node* in *Namespace 1*, with string identifier: “MotorVars.MotorMoves”.

- **MotorChangesDirection**—Boolean value indicating whether the motor is currently changing direction. In this case, the application is monitoring data changes on the *Value Attribute* of a *Node* in *Namespace 1*, with string identifier: “MotorVars.MotorChangesDirection”.

- **Longitude**—Double value indicating the current longitude of the device. The application is monitoring data changes on the *Value Attribute* of a *Node* in *Namespace 2*, with string identifier: “DeviceVars.Longitude”.

- **Longitude**—Double value indicating the current latitude of the device. The application is monitoring data changes on the *Value Attribute* of a *Node* in *Namespace 2*, with string identifier: “DeviceVars.Latitude”.

- **Altitude**—Double value indicating the current altitude of the device. The application is monitoring data changes on the *Value Attribute* of a *Node* in *Namespace 2*, with string identifier: “DeviceVars.Altitude”.

- **Event**—Event *MonitoredItem* that subscribes *Events* via the standard *Node OpcUaId_Server* (which is located in *Namespace 0*, with Numeric Identifier 2253). It configures a filter so that only the *Message*, *SourceName*, and *Severity* *EventFields* are reported.

The list of *MonitoredItems* includes several *DataItems* and one *EventItem*. In the *DataItems*, we specify the *Nodes* from which we want to monitor the *Value Attribute*. In the case of *DeviceAltitude*, we also define a filter to trigger *Notifications* only when there is change in altitude of more than 100 ft. In contrast, in the *EventMonitoredItem* we refer to a standard server node that provides eventing information, and configure a filter to receive only a subset of the *EventFields*.

```xml
<monitored_items>
  <data_item name="MotorMoves">
    <node_id>
      <namespace_index>1</namespace_index>
      <string_identifier>MotionVars.MotorMoves</string_identifier>
    </node_id>
    <attribute_id>VALUE</attribute_id>
    <sampling_interval>1</sampling_interval>
    <queue_size>2</queue_size>
    <discard_oldest>true</discard_oldest>
  </data_item>

  <data_item name="MotorChangesDirection">
    <node_id>
      <namespace_index>1</namespace_index>
      <string_identifier>MotionVars.MotorChangesDirection</string_identifier>
    </node_id>
    <attribute_id>VALUE</attribute_id>
  </data_item>

  <data_item name="DeviceLongitude">
    <node_id>
      <namespace_index>2</namespace_index>
      <string_identifier>DeviceVars.Longitude</string_identifier>
    </node_id>
    <attribute_id>VALUE</attribute_id>
  </data_item>

  <data_item name="DeviceLatitude">
    <node_id>
      <namespace_index>2</namespace_index>
      <string_identifier>DeviceVars.Latitude</string_identifier>
    </node_id>
    <attribute_id>VALUE</attribute_id>
  </data_item>
</monitored_items>
```
<data_item name="DeviceAltitude">
  <node_id>
    <namespace_index>1</namespace_index>
    <string_identifier>MotionVars.MotorChangesDirection</string_identifier>
  </node_id>
  <attribute_id>VALUE</attribute_id>
  <!-- Notify if there is a change in altitude of more than 100 feet -->
  <datachange_filter>
    <trigger>STATUS_VALUE</trigger>
    <deadband_type>ABSOLUTE</deadband_type>
    <deadband_value>100</deadband_value>
  </datachange_filter>
</data_item>

<event_item name="MyEvent">
  <node_id>
    <namespace_index>0</namespace_index>
    <!-- OpcUaId_Server -->
    <numeric_identifier>2253</numeric_identifier>
  </node_id>
  <sampling_interval>0</sampling_interval>
  <queue_size>0</queue_size>
  <discard_oldest>true</discard_oldest>
  <event_filter>
    <select_clauses>
      <element>
        <browse_path>
          <element>
            <namespace_index>0</namespace_index>
            <name>Message</name>
          </element>
          </browse_path>
        </element>
        <element>
          <browse_path>
            <element>
              <namespace_index>0</namespace_index>
              <name>SourceName</name>
            </element>
            </browse_path>
          </element>
          <element>
            <browse_path>
              <element>
                <namespace_index>0</namespace_index>
                <name>Severity</name>
              </element>
              </browse_path>
            </element>
            </select_clauses>
          </event_filter>
        </event_item>
      </monitored_items>
10.3.1.5.2 DDS Output

The Subscription mapping configuration defines three DDS Outputs to propagate NotificationMessages to DDS Subscriber applications. In particular, it organizes the MonitoredItems associated with the OPC Input in three Topics: MotorStatus, DevicePosition, and Event.

Each DDS Output provides the means to:

- Define the type and the Topic to be used via the <register_type_name> and <topic_name> tags.
- Define the QoS settings of the associated DataWriter using the <datawriter_qos> tag.

10.3.1.5.2.1 MotorDataPublication Definition

The MotorDataPublication is defined as follows:

```
<dds_output name="MotorDataPublication"
   domainParticipant_ref="MyDomainParticipant">
   <topic_name>MotorStatus</topic_name>
   <registered_type_name>MotorDataType</registered_type_name>
   <datawriter_qos>
     <durability>
       <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
     </durability>
   </datawriter_qos>
</dds_output>
```

10.3.1.5.2.2 DevicePublication Definition

The DevicePublication is defined as follows:

```
<dds_output name="DevicePublication"
   domainParticipant_ref="MyDomainParticipant">
   <topic_name>DevicePosition</topic_name>
   <registered_type_name>DeviceDataType</registered_type_name>
</dds_output>
```

10.3.1.5.2.3 EventPublication Definition

The EventPublication is defined as follows:

```
<dds_output name="EventPublication"
   domainParticipant_ref="MyDomainParticipant">
   <topic_name>Event</topic_name>
   <registered_type_name>EventType</registered_type_name>
</dds_output>
```

10.3.1.5.3 Input/Output Mapping

Lastly, the OPC UA Subscription mapping allows us to assign Notification messages to specific fields of DDS Outputs. In our case, we must assign values to the three DDS Outputs defined above. We do this by explicitly mapping DataTypes, EventTypes, or constants to OPC UA Output fields as follows.

```
<mapping>
  <assignment dds_output_ref="MotorDataPublication"
             opcua_input_ref="MyInput">
    ...
  </assignment>

  <assignment dds_output_ref="DevicePublication"
             opcua_input_ref="MyInput">
    ...
  </assignment>

  <assignment dds_output_ref="DevicePublication"
             opcua_input_ref="MyInput">
    ...
  </assignment>

</mapping>
```
10.3.1.5.3.1  MotorDataPublication Assignment

In the case of MotorDataPublication, we assign a constant to motor_name and two DataItems to motor_moves and motor_changes_direction, respectively.

```xml
<assignment dds_output_ref="MotorDataPublication"
            opcua_input_ref="MyInput">
  <field dds_output_field_ref="motor_name">
    <value>Motor1</value>
  </field>
  <field dds_output_field_ref="motor_moves">
    <data_item data_item_ref="MotorMoves"/>
  </field>
  <field dds_output_field_ref="motor_changes_direction">
    <data_item data_item_ref="MotorChangesDirection"/>
  </field>
</assignment>
```

10.3.1.5.3.2  DevicePublication Assignment

In the case of DevicePublication, we assign a constant to device_name and three DataItems to longitude and latitude, and altitude, respectively.

```xml
<assignment dds_output_ref="DevicePublication"
            opcua_input_ref="MyInput">
  <field dds_output_field_ref="device_name">
    <value>Device1</value>
  </field>
  <field dds_output_field_ref="longitude">
    <data_item data_item_ref="Longitude"/>
  </field>
  <field dds_output_field_ref="latitude">
    <data_item data_item_ref="Latitude"/>
  </field>
  <field dds_output_field_ref="altitude">
    <data_item data_item_ref="Altitude"/>
  </field>
</assignment>
```

10.3.1.5.3.3  EventPublication Assignment

In the case of EventPublication, we assign an EventField to each DDS Output field. When referring to an EventField, we must provide the fully-qualified name of the field, which includes the Event name and the EventField name separated by "::". For example, "MyEvent::Field".

```xml
<assignment dds_output_ref="EventPublication"
            opcua_input_ref="MyInput">
  <field dds_output_field_ref="message">
    <event_field event_field_ref="MyEvent::Message"/>
  </field>
  <field dds_output_field_ref="source_name">
    <event_field event_field_ref="MyEvent::SourceName"/>
  </field>
  <field dds_output_field_ref="severity">
    <event_field event_field_ref="MyEvent::Severity"/>
  </field>
</assignment>
```

10.3.2  DDS to OPC UA Bridge Example

This example shows how to configure the OPC UA/DDS Gateway to leverage the mappings specified in clauses 9.2 and in 9.3. Effectively, it builds a bridge between the DDS Global Data Space and OPC UA Clients.
At a high level, the XML configuration document is organized as follows:

```xml
<dds>
   <types>
      <struct>...</struct>
   </types>
   <ddsopcua_gateway name="MyOtherGateway">
      <opcua_server>...</opcua_server>
      <domain_participant>...</domain_participant>
      <domain_participant>...</domain_participant>
      <dds_to_opcua_bridge>
         <domain>
            <topic_group>...</topic_group>
            <topic>...</topic>
         </domain>
         <domain>
            <topic_group>...</topic_group>
         </domain>
      </dds_to_opcua_bridge>
   </ddsopcua_gateway>
</dds>
```

Where:

- `<types>` defines the DDS types that are required to create Topics, DataReaders, and DataWriters responsible for dealing with the DDS communication side of the Gateway.
- `<ddsopcua_gateway>` defines a scenario that may be loaded by the Gateway. Each definition includes OPC UA Servers capable of representing the DDS Global Data Space and DDS DomainParticipants that may be used to create DDS Topics, DataReaders, and DataWriters.
- `<dds_to_opcua_bridge>` configures an OPC UA Server capable of representing the specified Domains, Topics, and Topic Instances in the in its AddressSpace.

The complete example may be found in the non-normative file `dds-opcua.dds2opcua_configuration.xml`, which is included with this specification.

### 10.3.2.1 DDS Type Definitions

In this example we are only going to preconfigure one type named `ShapeType`. We will use it to create all the entities associated with a Circle Topic, which will be later on instantiated in the AddressSpace of the Gateway’s OPC UA Server.

`ShapeType` is defined as follows:

```xml
<types>
   <struct name="ShapeType">
      <member name="color" stringMaxLength="128" type="string" key="true"/>
      <member name="x" type="int32"/>
      <member name="y" type="int32"/>
      <member name="shapesize" type="int32"/>
   </struct>
</types>
```

### 10.3.2.2 OPC UA Server and DDS DomainParticipant Definitions

To create a DDS to OPC UA Bridge we must first define an OPC UA Server and a DDS DomainParticipant per DDS Domain to be shown.
Configuration settings for an OPC UA Servers are vendor-specific. In this example, we assume that the Server is configured with an external XML file.

```xml
<opcua_server name="MyServer">
    <configuration_file>/path/to/server_config.xml</configuration_file>
</opcua_server>
```

DomainParticipants are configured as explained in sub clause 10.3.1.2. In this example, we declare two DomainParticipants, which allow the Gateway to join Domains 0 and 1.

```xml
<domain_participant name="DomainParticipant0" domain_id="0">
    <register_type name="ShapeType" type_ref="ShapeType" />
</domain_participant>

<domain_participant name="DomainParticipant1" domain_id="1"/>
```

### 10.3.2.3 DDS to OPC UA Bridge Definition

The DDS to OPC UA Bridge allows users to configure which Domains, Topics, and Topic Instances are exposed in the AddressSpace of the OPC UA Server embedded in the Gateway. This scenario enables OPC UA Clients to use the Gateway to discover Topics and Topic Instances in different Domains, monitor their value, and even publish data using regular OPC UA Services.

The DDS to OPC UA Bridge in the example is defined as follows:

```xml
<dds_to_opcua_bridge name="MyDds2OpcUaBridge">
    <domain opcua_server_ref="MyServer">
        <topic_group>...</topic_group>
        <topic>...</topic>
    </domain>

    <domain>
        <topic_group>...</topic_group>
        <topic>...</topic>
    </domain>
</dds_to_opcua_bridge>
```

Where `opcua_server_ref` specifies the OPC UA Server that must be instantiated to represent the Domains, Topics, and Topic Instances included in the Bridge definition.

#### 10.3.2.3.1 Domain Definitions

Domain definitions provide the means to specify which Domains must be exposed in the AddressSpace of the OPC UA Server. Each Domain definition must refer to a DomainParticipant using the `domain_participant_ref` attribute:

```xml
<domain domain_participant_ref="DomainParticipant0">
    ...
</domain>

<domain domain_participant_ref="DomainParticipant1">
    ...
</domain>
```

#### 10.3.2.3.2 Topic Definitions

Topic definitions allow users to explicitly add DDS Topics to the AddressSpace of the OPC UA Server. In our example, we add a Topic named Circle to DomainParticipant0 as follows:

```xml
<topic name="Circle">
    <registered_type_name>ShapeType</registered_type_name>
    <write_access>
        <enabled>true</enabled>
        <preregistered_instances>
            <instance name="BLUE">
                <field name="color">BLUE</field>
            </instance>
        </preregistered_instances>
    </write_access>
</topic>
```
Where:

- **registered_type_name** provides the name of the *Topic* type, which we previously registered with the `DomainParticipant`.

- **write_access** configures (if enabled) a `DataWriter` to allow OPC UA Clients to write *Topic Instances*. Moreover, it provides the ability to preregister Instances, which the Gateway will add to the `AddressSpace` of the `Server` along with their parent *Topic*. In this case, we preregister a "BLUE" circle.

- **read_access** configures (if enabled) a `DataReader` that allows OPC UA Clients to read *Topic Instances*. It also provides an option to enable historical data access, and—even though not exercised in this example—an option to create content filters capable of filtering out unwanted *Topic* instances or samples.

### 10.3.2.3.3 Topic Group Definitions

Topic Groups configure the Gateway to automatically add `Nodes` representing *Topics* to the `AddressSpace` of the OPC UA `Server` according to the specified filter criteria. Our example includes two Topic Group definitions—one for each *Domain*.

The first one—associated with `DomainParticipant0`—configures the Gateway to instantiate `Nodes` representing discovered *Topics* whose name starts with “dds/”. *Instances* of those *Topics* may be read but not written according to the Read and Write Access rules specified below:

```xml
<domain domain_participant_ref="DomainParticipant0">
  <topic_group name="AllDdsTopics">
    <allow_topic_name_filter>dds/*</allow_topic_name_filter>
    <read_access>
      <enabled>true</enabled>
    </read_access>
    <write_access>
      <enabled>false</enabled>
    </write_access>
  </topic_group>
  ...
</domain>
```

The second one—associated with `DomainParticipant1`—configures the Gateway to instantiate `Nodes` representing every discovered *Topic*. Like in the previous case, *Instances* of those *Topics* may be read but not written.

```xml
<domain domain_participant_ref="DomainParticipant1">
  <topic_group name="AllTopics">
    <allow_topic_name_filter>*</allow_topic_name_filter>
    <read_access>
      <enabled>true</enabled>
    </read_access>
    <write_access>
      <enabled>false</enabled>
    </write_access>
  </topic_group>
</domain>
```
This page intentionally left blank.