



DPC: 24 / 30485904 DC

BSI Group Headquarters

389 Chiswick High Road London W4 4AL

Tel: +44 (0)20 8996 9000

Fax: +44 (0)20 8996 7400

www.bsigroup.com

Date: 02 February 2024

Origin: International

Latest date for receipt of comments: 27 March 2024

Project No. 2024/00104

Responsible committee: GEL/65/5 Devices and integration in enterprise systems

Interested committees:

Title: BS EN IEC 62541-6 OPC unified architecture

Part 6: Mappings

Please notify the secretary if you are aware of any keywords that might assist in classifying or identifying the standard or if the content of this standard

- i) has any issues related to 3rd party IPR, patent or copyright
- ii) affects other national standard(s)
- iii) requires additional national guidance or information

**WARNING: THIS IS A DRAFT AND MUST NOT BE REGARDED OR USED AS A BRITISH STANDARD.
THIS DRAFT IS NOT CURRENT BEYOND 27 March 2024**

This draft is issued to allow comments from interested parties; all comments will be given consideration prior to publication. No acknowledgement will normally be sent. **See overleaf for information on the submission of comments.**

No copying is allowed, in any form, without prior written permission from BSI except as permitted under the Copyright, Designs and Patent Act 1988 or for circulation within a nominating organization for briefing purposes. Electronic circulation is limited to dissemination by e-mail within such an organization by committee members.

Further copies of this draft may be purchased from BSI Shop <http://shop.bsigroup.com> or from BSI Customer Services, Tel: +44(0) 20 8996 9001 or email cservices@bsigroup.com. British, International and foreign standards are also available from BSI Customer Services.

Information on the co-operating organizations represented on the committees referenced above may be obtained from <http://standardsdevelopment.bsigroup.com>

Responsible Committee Secretary: Ms Charmaine Mukherjee (BSI)

Direct tel:

E-mail: charmaine.mukherjee@bsigroup.com

Introduction

This draft standard is based on national and international discussions. Your comments on this draft are invited and will assist in the preparation of the consequent standard.

For international standards, comments will be reviewed by the relevant UK national committee before sending the consensus UK vote and comments to the international committee, which will then decide appropriate action. If the international standard is approved, it is usual for the text to be published as a British Standard.

For national standards, comments will be reviewed by the relevant UK national committee and the resulting standards published as a British Standard.

UK Vote

Please indicate whether you consider the UK should submit a negative (with supporting technical reasons) or positive vote on this draft. Please indicate if you are aware of any reason why this draft standard should not be published as a British Standard.

Submission of Comments

- **Annotated drafts are not acceptable and will be rejected.**
- All comments should be submitted online at <https://standardsdevelopment.bsigroup.com/>. You will need to register in order to comment.

Template for comments and secretariat observations

Date: xx/xx/20xx	Document: ISO/DIS xxxx
------------------	------------------------

1	2	(3)	4	5	(6)	7
MB	Clause No./ Subclause No./Annex (e.g. 3.1)	Paragraph/Figure/ Table/Note	Type of comment	Commend (justification for change) by the MB	Proposed change by the MB	Secretariat observations on each comment submitted
	3.1	Definition 1	ed	Definition is ambiguous and needs clarifying.	Amend to read '...so that the mains connector to which no connection...'	
	6.4	Paragraph 2	te	The use of the UV photometer as an alternative cannot be supported as serious problems have been encountered in its use in the UK.	Delete reference to UV photometer.	

PROJECT NUMBER:

IEC 62541-6 ED4

DATE OF CIRCULATION:

2024-02-02

CLOSING DATE FOR VOTING:

2024-04-26

SUPERSEDES DOCUMENTS:

65E/978/RR

IEC SC 65E : DEVICES AND INTEGRATION IN ENTERPRISE SYSTEMS

SECRETARIAT:

United States of America

SECRETARY:

Mr Donald (Bob) Lattimer

OF INTEREST TO THE FOLLOWING COMMITTEES:

PROPOSED HORIZONTAL STANDARD:



Other TC/SCs are requested to indicate their interest, if any, in this CDV to the secretary.

FUNCTIONS CONCERNED:

☐ EMC☐ ENVIRONMENT☐ QUALITY ASSURANCE☐ SAFETY☒ SUBMITTED FOR CENELEC PARALLEL VOTING☐ NOT SUBMITTED FOR CENELEC PARALLEL VOTING**Attention IEC-CENELEC parallel voting**

The attention of IEC National Committees, members of CENELEC, is drawn to the fact that this Committee Draft for Vote (CDV) is submitted for parallel voting.

The CENELEC members are invited to vote through the CENELEC online voting system.

This document is still under study and subject to change. It should not be used for reference purposes.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Recipients of this document are invited to submit, with their comments, notification of any relevant "In Some Countries" clauses to be included should this proposal proceed. Recipients are reminded that the CDV stage is the final stage for submitting ISC clauses. (SEE AC/22/2007 OR NEW GUIDANCE DOC).

TITLE:

OPC Unified Architecture - Part 6: Mappings

PROPOSED STABILITY DATE: 2026

NOTE FROM TC/SC OFFICERS:

Copyright © 2024 International Electrotechnical Commission, IEC. All rights reserved. It is permitted to download this electronic file, to make a copy and to print out the content for the sole purpose of preparing National Committee positions. You may not copy or "mirror" the file or printed version of the document, or any part of it, for any other purpose without permission in writing from IEC.

CONTENTS

	Page
FIGURES	iv
TABLES	iv
FOREWORD	viii
1. Scope	1
2. Normative references	1
3. Terms, definitions and abbreviated terms	3
3.1. Terms and definitions	3
3.2. Abbreviated terms	4
4. Overview	5
5. Data encoding	6
5.1. General	6
5.1.1. Overview	6
5.1.2. Built-in Types	6
5.1.3. Guid	7
5.1.4. DateTime	7
5.1.5. ByteString	7
5.1.6. Number, Integer and UInteger	8
5.1.7. Structures and Unions	8
5.1.8. ExtensionObject	8
5.1.9. Variant	9
5.1.10. Decimal	9
5.1.11. Null, Empty and Zero-Length Arrays	10
5.1.12. QualifiedName, NodeId and ExpandedNodeId String Encoding	10
5.1.13. Name Encoding Rules	12
5.2. OPC UA Binary	12
5.2.1. General	12
5.2.2. Built-in Types	13
5.2.3. Decimal	21
5.2.4. Enumerations	21
5.2.5. Arrays	21
5.2.6. Structures	22
5.2.7. Structures with optional fields	25
5.2.8. Unions	26
5.2.9. Messages	28
5.3. OPC UA XML	28
5.3.1. Built-in Types	28
5.3.2. Decimal	33
5.3.3. Enumerations	34
5.3.4. Arrays	34
5.3.5. Structures	35
5.3.6. Structures with optional fields	35
5.3.7. Unions	35
5.3.8. Messages	36
5.4. OPC UA JSON	36
5.4.1. General	36

49	5.4.2.	Built-in Types	37
50	5.4.3.	Decimal	41
51	5.4.4.	Enumerations	41
52	5.4.5.	Arrays	42
53	5.4.6.	Structures	42
54	5.4.7.	Structures with optional fields	43
55	5.4.8.	Unions	44
56	5.4.9.	Messages	44
57	6.	Message Security Protocols	44
58	6.1.	Certificates	46
59	6.1.1.	General	46
60	6.1.2.	Application Instance Certificate	46
61	6.1.3.	User Certificates	47
62	6.1.4.	Issuer (CA) Certificates	48
63	6.1.5.	Certificate Revocation List (CRL)	48
64	6.1.6.	Certificate Chains	48
65	6.2.	Time synchronization	49
66	6.3.	UTC and International Atomic Time (TAI)	49
67	6.4.	Issued User Identity Tokens	49
68	6.4.1.	Kerberos	49
69	6.4.2.	JSON Web Token (JWT)	49
70	6.4.3.	OAuth2	50
71	6.5.	WS Secure Conversation	52
72	6.6.	OPC UA Secure Conversation	52
73	6.6.1.	Overview	52
74	6.6.2.	MessageChunk structure	52
75	6.6.3.	MessageChunks and error handling	56
76	6.6.4.	Establishing a SecureChannel	57
77	6.6.5.	Deriving keys	58
78	6.6.6.	Verifying Message Security	59
79	6.7.	Elliptic Curve Cryptography (ECC)	60
80	6.7.1.	Secure Channel Handshake	60
81	6.7.2.	UserIdentityToken Encryption	64
82	6.7.3.	ECC Encrypted Secret	65
83	7.	Transport Protocols	66
84	7.1.	OPC UA Connection Protocol	66
85	7.1.1.	Overview	66
86	7.1.2.	Message structure	66
87	7.1.3.	Establishing a connection	69
88	7.1.4.	Closing a connection	71
89	7.1.5.	Error handling	71
90	7.2.	OPC UA TCP	72
91	7.3.	SOAP/HTTP	72
92	7.4.	OPC UA HTTPS	73
93	7.4.1.	Overview	73
94	7.4.2.	Session-less Services	74
95	7.4.3.	XML Encoding	74
96	7.4.4.	OPC UA Binary Encoding	75
97	7.4.5.	JSON Encoding	76

98	7.5. WebSockets	76
99	7.5.1. Overview	76
100	7.5.2. Protocol Mapping	77
101	7.5.3. Security	77
102	1.1 Well known addresses	78
103	8. Normative Contracts	78
104	8.1. OPC Binary Schema	78
105	8.2. XML Schema and WSDL	78
106	8.3. Information Model Schema	79
107	8.4. Formal definition of UA Information Model	79
108	8.5. Constants	79
109	8.6. DataType encoding	79
110	8.7. Security configuration	79
111	Annex A (normative) Constants	80
112	A.1 Attribute Ids	80
113	A.2 Status Codes	80
114	A.3 Numeric Node Ids	80
115	A.4 Media Types	81
116	Annex B (normative) OPC UA NodeSet	82
117	Annex C (normative) Type declarations for the OPC UA native Mapping	83
118	Annex D (normative) WSDL for the XML Mapping	84
119	D.1 XML Schema	84
120	D.2 WDSL Port Types	84
121	D.3 WSDL Bindings	84
122	Annex E (informative) Security settings management	85
123	E.1 Overview	85
124	E.2 SecuredApplication	86
125	E.3 CertificateIdentifier	90
126	E.4 CertificateStoreIdentifier	91
127	E.5 CertificateList	92
128	E.6 CertificateValidationOptions	92
129	Annex F (normative) Information Model XML Schema	94
130	F.1 Overview	94
131	F.2 UANodeSet	94
132	F.3 UANode	96
133	F.4 Reference	98
134	F.5 RolePermission	98
135	F.6 UAType	98
136	F.7 UAInstance	99
137	F.8 UAVariable	99
138	F.9 UAMethod	100
139	F.10 TranslationType	101
140	F.11 UADatatype	101
141	F.12 DataTypeDefinition	102
142	F.13 DataTypeField	103
143	F.14 Variant	104
144	F.15 Example	105
145	F.16 UANodeSetChanges	107

146	F.17	NodesToAdd	107
147	F.18	ReferencesToChange	108
148	F.19	ReferenceToChange	108
149	F.20	NodesToDelete	108
150	F.21	NodeToDelete	109
151	F.22	UANodeSetChangesStatus	109
152	F.23	NodeSetStatusList	109
153	F.24	NodeSetStatus	110

154

155

156

FIGURES

157

158	Figure 1 – The OPC UA Stack Overview	5
159	Figure 2 – Encoding Integers in a binary stream	13
160	Figure 3 – Encoding Floating Points in a binary stream	13
161	Figure 4 – Encoding Strings in a binary stream	14
162	Figure 5 – Encoding Guids in a binary stream	15
163	Figure 6 – Encoding XmlElement in a binary stream	15
164	Figure 7 – A String NodeId	16
165	Figure 8 – A Two Byte NodeId	16
166	Figure 9 – A Four Byte NodeId	17
167	Figure 10 – Security handshake when Creating a Session	45
168	Figure 11 – MessageChunk for Unauthenticated Encryption Algorithms	52
169	Figure 12 – MessageChunk for Authenticated Encryption Algorithms	53
170	Figure 13 – ECC Key Negotiation	60
171	Figure 14 – Signing and Encryption with Authenticated Encryption	63
172	Figure 15 – Signing and Encryption with Unauthenticated Encryption	63
173	Figure 16 – ECC CreateSession/ActivateSession Handshake	64
174	Figure 17 – OPC UA Connection Protocol Message structure	66
175	Figure 18 – Client initiated OPC UA Connection Protocol connection	70
176	Figure 19 – Server initiated OPC UA Connection Protocol connection	70
177	Figure 20 – Closing a OPC UA Connection Protocol connection	71
178	Figure 21 – Scenarios for the HTTPS Transport	73
179	Figure 22 – Setting up Communication over a WebSocket	77

180

181

182

TABLES

183	Table 1 – Built-in Data Types	6
184	Table 2 – Guid structure	7
185	Table 3 – Layout of Decimal	9
186	Table 4 – Additional ABNF Core Rules	10
187	Table 5 – ABNF Description for a NodeId	10
188	Table 6 – ABNF Description for a ExpandedNodeId	11
189	Table 7 – ABNF Description for a QualifiedName	11

190	Table 10 – Examples of XML Encoded Names	12
191	Table 11 – Supported Floating Point Types.....	13
192	Table 12 – NodeId components	15
193	Table 13 – NodeId DataEncoding values.....	16
194	Table 14 – Standard NodeId Binary DataEncoding	16
195	Table 15 – Two Byte NodeId Binary DataEncoding.....	16
196	Table 16 – Four Byte NodeId Binary DataEncoding	17
197	Table 17 – ExpandedNodeId Binary DataEncoding.....	17
198	Table 18 – DiagnosticInfo Binary DataEncoding	18
199	Table 19 – QualifiedName Binary DataEncoding	18
200	Table 20 – LocalizedText Binary DataEncoding.....	19
201	Table 21 – Extension Object Binary DataEncoding	19
202	Table 22 – Variant Binary DataEncoding.....	20
203	Table 23 – Data Value Binary DataEncoding	21
204	Table 24 – Inline Matrix DataEncoding.....	22
205	Table 25 – Sample OPC UA Binary Encoded structure	23
206	Table 26 – DataTypeDefinition for “Type1” from Sample.....	24
207	Table 27 –DataTypeDefinition for “Type2” from Sample.....	25
208	Table 28 – Sample OPC UA Binary Encoded Structure with optional fields	26
209	Table 29 – Sample OPC UA Binary Encoded Structure	27
210	Table 30 – XML Data Type Mappings for Integers	29
211	Table 31 – XML Data Type Mappings for Floating Points.....	29
212	Table 32 – Components of Enumeration	34
213	Table 33 – JSON Object Definition for a NodeId.....	38
214	Table 34 – JSON Object Definition for an ExpandedNodeId.....	38
215	Table 35 – JSON Object Definition for a StatusCode.....	39
216	Table 36 – JSON Object Definition for a DiagnosticInfo	39
217	Table 37 – JSON Object Definition for a QualifiedName	40
218	Table 38 – JSON Object Definition for a LocalizedText	40
219	Table 39 – JSON Object Definition for an ExtensionObject.....	40
220	Table 40 – JSON Object Definition for a Variant.....	41
221	Table 41 – JSON Object Definition for a DataValue.....	41
222	Table 42 – JSON Object Definition for a Decimal	41
223	Table 43 – JSON Encoding Rules for Structures	42
224	Table 44 – JSON Object Definition for a Structures with Optional Fields	43
225	Table 45 – JSON Encoding Rules for Structures with Optional Fields	43
226	Table 46 – JSON Object Definition for a Union.....	44
227	Table 47 – SecurityPolicy	45
228	Table 48 – Application Instance Certificate	47
229	Table 49 – User Certificate	48
230	Table 50 – Issuer Certificate.....	48
231	Table 51 – Certificate Revocation List Extensions	48
232	Table 52 – JWT UserTokenPolicy	50

233	Table 53 – JWT IssuerEndpointUrl Definition	50
234	Table 54 – Access Token Claims	51
235	Table 55 – OPC UA Secure Conversation Message Header	53
236	Table 56 – Asymmetric algorithm Security header.....	54
237	Table 57 – Symmetric algorithm Security header.....	55
238	Table 58 – Sequence header	55
239	Table 59 – Message Footer for Unauthenticated Encryption Algorithms	56
240	Table 60 – Message Footer for Authenticated Encryption Algorithms	56
241	Table 61 – OPC UA Secure Conversation Message abort body	57
242	Table 62 – OPC UA Secure Conversation OpenSecureChannel Service	57
243	Table 63 – PRF inputs for RSA based SecurityPolicies	59
244	Table 64 – Cryptography key generation parameters.....	59
245	Table 65 – Deriving Client Keys from Keying Material	62
246	Table 66 – Deriving Server Keys from Keying Material	62
247	Table 67 – Creating a Mask for the Initialization Vector	64
248	Table 68 – Additional Header Key Names	65
249	Table 69 – Deriving Keys from Keying Material	66
250	Table 70 – OPC UA Connection Protocol Message header.....	67
251	Table 71 – OPC UA Connection Protocol Hello Message.....	67
252	Table 72 – OPC UA Connection Protocol Acknowledge Message	68
253	Table 73 – OPC UA Connection Protocol Error Message	68
254	Table 74 – OPC UA Connection Protocol ReverseHello Message	69
255	Table 75 – Client and Server Handshake during Reverse Connect	70
256	Table 76 – OPC UA Connection Protocol error codes.....	72
257	Table 77 – WebSocket Protocols Mappings	77
258	Table 78 – Well known addresses for Local Discovery Servers.....	78
259	Table A.1 – Identifiers assigned to Attributes	80
260	Table A.2 – Media Types Assigned to OPC UA Document Formats	81
261	Table E.1 – SecuredApplication	87
262	Table E.2 – CertificateIdentifier	90
263	Table E.3 – Structured directory store.....	91
264	Table E.4 – CertificateStoreIdentifier	92
265	Table E.5 – CertificateList	92
266	Table E.6 – CertificateValidationOptions.....	93
267	Table F.1 – UANodeSet.....	95
268	Table F.2 – UANode.....	97
269	Table F.3 – Reference.....	98
270	Table F.4 – RolePermission.....	98
271	Table F.5 – UANodeSet Type Nodes	99
272	Table F.6 – UANodeSet Instance Nodes	99
273	Table F.7 – UAInstance	99
274	Table F.8 – UAVariable	100
275	Table F.9 – UAMethod	100

276	Table F.10 – TranslationType	101
277	Table F.11 – UADDataType	102
278	Table F.12 – DataTypeDefinition	102
279	Table F.13 – StructureType Enumeration Mapping	103
280	Table F.14 – DataTypeField	104
281	Table F.15 – UANodeSetChanges	107
282	Table F.16 – NodesToAdd	108
283	Table F.17 – ReferencesToChange	108
284	Table F.18 – ReferencesToChange	108
285	Table F.19 – NodesToDelete	109
286	Table F.20 – ReferencesToChange	109
287	Table F.21 – UANodeSetChangesStatus.....	109
288	Table F.22 – NodeSetStatusList	110
289	Table F.23 – NodeSetStatus.....	110

290

291

292

INTERNATIONAL ELECTROTECHNICAL COMMISSION

OPC UNIFIED ARCHITECTURE –

Part 6: Mappings

FOREWORD

- 1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
- 2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
- 3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
- 4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
- 5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
- 6) All users should ensure that they have the latest edition of this publication.
- 7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
- 8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
- 9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 62541-6 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

This fourth edition cancels and replaces the third edition published in 2020. This edition constitutes a technical revision.

This edition includes the following significant technical changes with respect to the previous edition:

- a) Added support for ECC to UA Secure Conversation
- b) Added use of the AuthorityKeyIdentifier extension in Certificate Revocation Lists.
- c) Enhanced JSON mapping of Unions.
- d) Added Decimal data type encoding.
- e) Specified ECC keyUsage rules.
- f) Added Media assigned by IANA to UANodeSet definition.
- g) Added requirements for user and issuer Certificates.
- h) Added rules which specify what happens when DateTime precision is lost.
- i) Added rules to allow for the truncation of strings containing embedded nulls.

- j) Defined a normative string representation for NodeId, ExpandedNodeId and QualifiedName for JSON mapping.
- k) Require that TAI times be converted to UTC.
- l) Allow Symbol to be omitted if unknown in JSON encoding.
- m) Added fields needed to support RolePermissions to the UANodeSet.

The text of this International Standard is based on the following documents:

CDV	Report on voting
65E/XX/CDV	65E/XX/RVC

Full information on the voting for the approval of this International Standard can be found in the report on voting indicated in the above table.

This document has been drafted in accordance with the ISO/IEC Directives, Part 2.

Throughout this document and the other parts of the IEC 62541 series, certain document conventions are used:

Italics are used to denote a defined term or definition that appears in the "Terms and definition" clause in one of the parts of the IEC 62541 series.

Italics are also used to denote the name of a service input or output parameter or the name of a structure or element of a structure that are usually defined in tables.

The *italicized terms and names* are, with a few exceptions, written in camel-case (the practice of writing compound words or phrases in which the elements are joined without spaces, with each element's initial letter capitalized within the compound). For example, the defined term is *AddressSpace* instead of Address Space. This makes it easier to understand that there is a single definition for *AddressSpace*, not separate definitions for Address and Space.

A list of all parts of the IEC 62541 series, published under the general title *OPC Unified Architecture*, can be found on the IEC website.

The committee has decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under "<http://webstore.iec.ch>" in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

OPC Unified Architecture Specification

Part 6: Mappings

1. Scope

This part of OPC Unified Architecture (OPC UA) specifies the mapping between the security model described in IEC 62541-2, the abstract service definitions specified in IEC 62541-4, the data structures defined in IEC 62541-5 and the physical network protocols that can be used to implement the OPC UA specification.

2. Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 62541-1, *OPC Unified Architecture - Part 1: Overview and Concepts*

IEC 62541-2, *OPC Unified Architecture - Part 2: Security Model*

IEC 62541-3, *OPC Unified Architecture - Part 3: Address Space Model*

IEC 62541-4, *OPC Unified Architecture - Part 4: Services*

IEC 62541-5, *OPC Unified Architecture - Part 5: Information Model*

IEC 62541-6, *OPC Unified Architecture - Part 6: Mappings*

IEC 62541-7, *OPC Unified Architecture - Part 7: Profiles*

IEC 62541-9, *OPC Unified Architecture - Part 9: Alarms and Conditions*

IEC 62541-12, *OPC Unified Architecture - Part 12: Discovery and Global Services*

IEC 62541-18, *OPC Unified Architecture - Part 18: Role-Based Security*

IEC 62541-21, *OPC Unified Architecture - Part 21: Device Onboarding*
OPC Unified Architecture - Part 12: Discovery and Global Services

XML Schema Part 2: XML Schema Part 2: Datatypes

<http://www.w3.org/TR/xmlschema-2/>

SOAP Part 1, SOAP Version 1.2 Part 1: Messaging Framework

<http://www.w3.org/TR/soap12-part1/>

WS Addressing, Web Services Addressing (WS-Addressing)

<http://www.w3.org/Submission/ws-addressing/>

TLS, RFC 8446 – The Transport Layer Security (TLS) Protocol Version 1.3

<https://datatracker.ietf.org/doc/html/rfc8446>

X.509 v3, ISO/IEC 9594-8 (ITU-T Rec. X.509), Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks

<https://www.iso.org/standard/72557.html>

HTTP, Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing

<https://datatracker.ietf.org/doc/html/rfc7230>

HTTPS, HTTP Over TLS

<http://www.ietf.org/rfc/rfc2818.txt>

Base64, The Base16, Base32, and Base64 Data Encodings

423 <https://datatracker.ietf.org/doc/html/rfc4648>

424 X690, ISO/IEC 8825-1 (ITU-T Rec. X.690), Information technology – ASN.1 encoding rules:
425 Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and
426 Distinguished Encoding Rules (DER)

427 <https://www.iso.org/standard/35688.html>

428 X200, ISO/IEC 7498-1 (ITU-T Rec. X.200), Information technology – Open Systems
429 Interconnection – Basic Reference Model: The Basic Model

430 <https://www.iso.org/standard/20269.html>

431 IEEE 754, 60559-2020 - ISO/IEC/IEEE International Standard - Standard for Floating-Point
432 Arithmetic

433 <https://ieeexplore.ieee.org/document/9091348>

434 HMAC, Keyed-Hashing for Message Authentication

435 <http://www.ietf.org/rfc/rfc2104.txt>

436 PKCS #1, RSA Cryptography Specifications Version 2.2

437 <https://datatracker.ietf.org/doc/html/rfc8017>

438 PKCS #12, Personal Information Exchange Syntax v1.1

439 <https://datatracker.ietf.org/doc/html/rfc7292>

440 FIPS 180-4, Secure Hash Standard (SHS)

441 <https://csrc.nist.gov/publications/detail/fips/180/4/final>

442 FIPS 197, Advanced Encryption Standard (AES)

443 <https://www.nist.gov/publications/advanced-encryption-standard-aes>

444 UTF-8, UTF-8, a transformation format of ISO 10646

445 <https://datatracker.ietf.org/doc/html/rfc3629>

446 RFC 5280, Internet X.509 Public Key Infrastructure Certificate
447 and Certificate Revocation List (CRL) Profile

448 <https://datatracker.ietf.org/doc/html/rfc5280>

449 RFC 4514, LDAP: String Representation of Distinguished Names

450 <https://datatracker.ietf.org/doc/html/rfc4514>

451 NTP, Network Time Protocol Version 4: Protocol and Algorithms Specification

452 <https://datatracker.ietf.org/doc/html/rfc5905>

453 RFC 3986, Uniform Resource Identifier (URI): Generic Syntax

454 <https://datatracker.ietf.org/doc/html/rfc3986>

455 RFC 8141, Uniform Resource Names (URNs)

456 <https://datatracker.ietf.org/doc/html/rfc8141>

457 RFC 6455, The WebSocket Protocol

458 <https://datatracker.ietf.org/doc/html/rfc6455>

459 RFC 8259, The JavaScript Object Notation (JSON) Data Interchange Format

460 <https://datatracker.ietf.org/doc/html/rfc8259>

461 RFC 7523, JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and
462 Authorization Grants

463 <https://datatracker.ietf.org/doc/html/rfc7523>

464 RFC 5392, The OAuth 2.0 Authorization Framework

465 <https://datatracker.ietf.org/doc/html/rfc6749>

OpenID-Core, OpenID Connect Core 1.0

http://openid.net/specs/openid-connect-core-1_0.html

OpenID-Discovery, OpenID Connect Discovery 1.0

https://openid.net/specs/openid-connect-discovery-1_0.html

RFC 6960, X.509 Internet Public Key Infrastructure – Online Certificate Status Protocol – OCSP

<https://datatracker.ietf.org/doc/html/rfc6960>

RFC 5869, HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

<https://datatracker.ietf.org/doc/html/rfc5869>

RFC 8422, Elliptic Curve Cryptography (ECC) Cipher Suites

<https://datatracker.ietf.org/doc/html/rfc8422>

RFC 6750, The OAuth 2.0 Authorization Framework: Bearer Token Usage

<https://datatracker.ietf.org/doc/html/rfc6750>

ISO 8601-1, Date and time - Representations for information interchange - Part 1: Basic rules

<https://www.iso.org/standard/70907.html>

RFC 5234, Augmented BNF for Syntax Specifications: ABNF

<https://datatracker.ietf.org/doc/html/rfc5234>

RFC 4151, The 'tag' URI Scheme

<https://datatracker.ietf.org/doc/html/rfc4151>

SemVer, Semantic Versioning 2.0.0

<https://semver.org/spec/v2.0.0.html>

3. Terms, definitions and abbreviated terms

3.1. Terms and definitions

For the purposes of this document the terms and definitions given in IEC 62541-1, IEC 62541-2 and IEC 62541-3 as well as the following apply.

3.1.1.

CertificateDigest

short identifier used to uniquely identify an X.509v3 *Certificate*.

Note 1 to entry: This is the SHA1 hash of DER encoded form of the *Certificate*.

3.1.2.

DataEncoding

way to serialize OPC UA *Messages* and data structures.

3.1.3.

DefaultValue

a value that is used when no other value is known or available.

3.1.4.

DevelopmentPlatform

suite of tools and/or programming languages used to create software.

3.1.5.

Mapping

specification on how to implement an OPC UA feature with a specific technology.

Note 1 to entry: For example, the OPC UA Binary Encoding is a *Mapping* that specifies how to serialize OPC UA data structures as sequences of bytes.

3.1.6.**OctetString**

a sequence of zero or more octets without any length prefix.

Note 1 to entry: an OctetString is not defined as a *DataType* and only used in this document when a *ByteString* is not appropriate.

3.1.7.**SecurityProtocol**

Protocol which ensures the integrity and privacy of UA *Messages* that are exchanged between OPC UA applications

3.1.8.**StackProfile**

combination of *DataEncodings*, *SecurityProtocol* and *TransportProtocol Mappings*

Note 1 to entry: OPC UA applications implement one or more *StackProfiles* and can only communicate with OPC UA applications that support a *StackProfile* that they support.

3.1.9.**TransportConnection**

full-duplex communication link established between OPC UA applications.

Note 1 to entry: A TCP/IP socket is an example of a *TransportConnection*.

3.1.10.**TransportProtocol**

way to exchange serialized OPC UA *Messages* between OPC UA applications

3.2. Abbreviated terms

AEAD	Authenticated Encryption with Associated Data
API	Application Programming Interface
ASN.1	Abstract Syntax Notation #1 (used in X690)
CSV	Comma Separated Value (File Format)
DNS	Domain Name Service
ECC	Elliptic Curve Cryptography
HTTP	Hypertext Transfer Protocol
HTTPS	Secure Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IPSec	Internet Protocol Security
NAT	Network Address Translation
OID	Object Identifier (used with ASN.1)
PRF	Pseudo Random Function
RSA	Rivest, Shamir and Adleman [Public Key Encryption System]
SHA1	Secure Hash Algorithm
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer (Defined in TLS)
TCP	Transmission Control Protocol
TLS	Transport Layer Security (Defined in TLS)
UA	Unified Architecture
UACP	OPC UA Connection Protocol
UASC	OPC UA Secure Conversation
WS-*	XML Web Services Specifications
XML	Extensible Markup Language

4. Overview

Other parts of the OPC 10000 series are written to be independent of the technology used for implementation. This approach means OPC UA is a flexible specification that will continue to be applicable as technology evolves. On the other hand, this approach means that it is not possible to build an OPC UA application with the information contained in IEC 62541-1 through to IEC 62541-5 because important implementation details have been left out.

This document defines *Mappings* between the abstract specifications and technologies that can be used to implement them. The *Mappings* are organized into three groups: *DataEncodings*, *SecurityProtocols* and *TransportProtocols*. Different *Mappings* are combined together to create *StackProfiles*. All OPC UA applications shall implement at least one *StackProfile* and can only communicate with other OPC UA applications that implement the same *StackProfile*.

This document defines the *DataEncodings* in Clause 0, the *SecurityProtocols* in 5.4 and the *TransportProtocols* in 6.7.6. The *StackProfiles* are defined in IEC 62541-7.

All communication between OPC UA applications is based on the exchange of *Messages*. The parameters contained in the *Messages* are defined in IEC 62541-4; however, their format is specified by the *DataEncoding* and *TransportProtocol*. For this reason, each *Message* defined in IEC 62541-4 shall have a normative description which specifies exactly what shall be put on the wire. The normative descriptions are defined in the annexes.

A *Stack* is a collection of software libraries that implement one or more *StackProfiles*. The interface between an OPC UA application and the *Stack* is a non-normative API which hides the details of the *Stack* implementation. An API depends on a specific *DevelopmentPlatform*. Note that the datatypes exposed in the API for a *DevelopmentPlatform* may not match the datatypes defined by the specification because of limitations of the *DevelopmentPlatform*. For example, Java Programming Language does not support an unsigned integer which means that any API will need to map unsigned integers onto a signed integer type.

Figure 1 illustrates the relationships between the different concepts defined in this document.

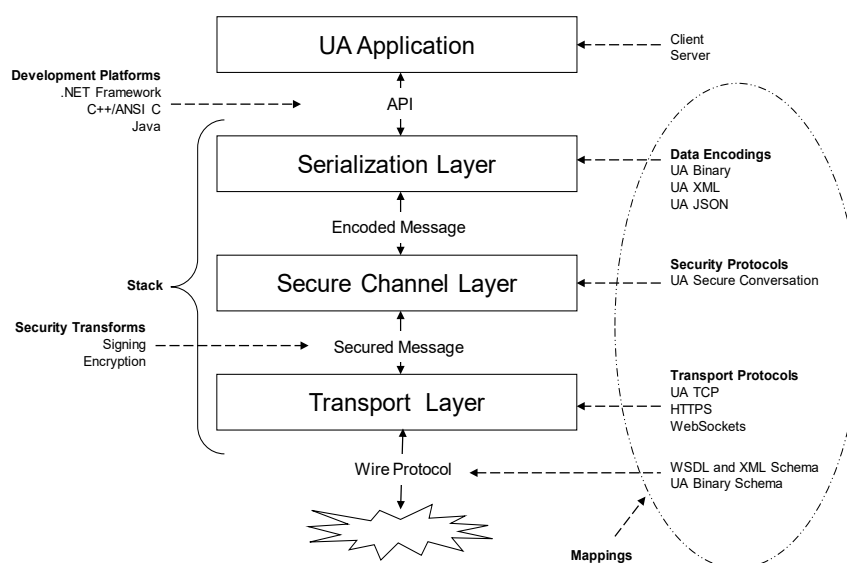


Figure 1 – The OPC UA Stack Overview

The layers described in this specification do not correspond to layers in the OSI 7-layer model [X200]. Each OPC UA *StackProfile* should be treated as a single Layer 7 (application) protocol that is built on an existing Layer 5, 6 or 7 protocol such as TCP/IP, TLS or HTTP. The *SecureChannel* layer is always present even if the *SecurityMode* is *None*. In this situation, no security is applied but the *SecurityProtocol* implementation shall maintain a logical channel with a unique identifier. Users and administrators are expected to understand that a *SecureChannel*

with *SecurityMode* set to *None* cannot be trusted unless the application is operating on a physically secure network or a low-level protocol such as IPSec is being used.

5. Data encoding

5.1. General

5.1.1. Overview

This document defines three *DataEncodings*: OPC UA Binary, OPC UA XML and OPC UA JSON. It describes how to construct *Messages* using each of these encodings.

5.1.2. Built-in Types

All OPC UA *DataEncodings* are based on rules that are defined for a standard set of built-in types. These built-in types are then used to construct structures, arrays and *Messages*. The built-in types are described in Table 1.

Table 1 – Built-in Data Types

ID	Name	Nullable	Default	Description
1	Boolean	No	false	A two-state logical value (true or false).
2	SByte	No	0	An integer value between –128 and 127 inclusive.
3	Byte	No	0	An integer value between 0 and 255 inclusive.
4	Int16	No	0	An integer value between –32 768 and 32 767 inclusive.
5	UInt16	No	0	An integer value between 0 and 65 535 inclusive.
6	Int32	No	0	An integer value between –2 147 483 648 and 2 147 483 647 inclusive.
7	UInt32	No	0	An integer value between 0 and 4 294 967 295 inclusive.
8	Int64	No	0	An integer value between –9 223 372 036 854 775 808 and 9 223 372 036 854 775 807 inclusive.
9	UInt64	No	0	An integer value between 0 and 18 446 744 073 709 551 615 inclusive.
10	Float	No	0	An IEEE single precision (32 bit) floating point value.
11	Double	No	0	An IEEE double precision (64 bit) floating point value.
12	String	Yes	null	A sequence of Unicode characters.
13	DateTime	Yes	DateTime.MinValue (see 5.1.4)	An instance in time.
14	Guid	Yes	All zeros	A 16-byte value that can be used as a globally unique identifier.
15	ByteString	Yes	null	A sequence of octets.
16	XmlElement	Yes	null	A sequence of Unicode characters that is an XML element.
17	NodeId	Yes	All fields set to default.	An identifier for a node in the address space of an OPC UA Server.
18	ExpandedNodeId	Yes	All fields set to default.	A NodeId that allows the namespace URI to be specified instead of an index.
19	StatusCode	No	Good	A numeric identifier for an error or condition that is associated with a value or an operation.
20	QualifiedName	Yes	All fields set to default.	A name qualified by a namespace.
21	LocalizedText	Yes	All fields set to default.	Human readable text with an optional locale identifier.
22	ExtensionObject	Yes	All fields set to default.	A structure that contains an application specific data type that may not be recognized by the receiver.
23	DataValue	Yes	All fields set to default.	A data value with an associated status code and timestamps.
24	Variant	Yes	Null	A union of all of the types specified above.
25	DiagnosticInfo	Yes	No fields specified.	A structure that contains detailed error and diagnostic information associated with a StatusCode.

Most of these data types are the same as the abstract types defined in IEC 62541-3 and IEC 62541-4. However, the *ExtensionObject* and *Variant* types are defined in this document. In addition, this document defines a representation for the *Guid* type defined in IEC 62541-3.

The Nullable column indicates whether a 'null' value exists for the *DataType* in all *DataEncodings*. A 'null' value is a value that is equivalent 'no value specified'. A nullable type with a default value means the default value shall be interpreted equivalent to a null.

The Default column specifies the default value for the type if a default value is needed. The default value for all arrays is 'null'.

5.1.3. Guid

A *Guid* is a 16-byte globally unique identifier with the layout shown in Table 2.

Table 2 – Guid structure

Component	Data Type
Data1	UInt32
Data2	UInt16
Data3	UInt16
Data4	Byte [8]

Guid values may be represented as a string in this form:

```
<Data1>-<Data2>-<Data3>-<Data4[0:1]>-<Data4[2:7]>
```

where Data1 is 8 characters wide, Data2 and Data3 are 4 characters wide and each *Byte* in Data4 is 2 characters wide. Each value is formatted as a hexadecimal number with padded zeros. A typical *Guid* value would look like this when formatted as a string:

```
C496578A-0DFE-4B8F-870A-745238C6AEAE
```

5.1.4. DateTime

DateTime values have different ranges on different *DevelopmentPlatforms*. To ensure interoperability two named values are defined:

"DateTime.MinValue" is the earliest value that can be represented;

"DateTime.MaxValue" is the latest value that can be represented.

If the range supported by the *DataEncoding* is outside of the range supported by a *DevelopmentPlatform* then decoders shall replace any below range values with DateTime.MinValue and any above range values with DateTime.MaxValue for the *DevelopmentPlatform*.

If the range supported by a *DevelopmentPlatform* is outside of the range supported by a *DataEncoding* then encoders shall replace any below range values with DateTime.MinValue and any above range values with DateTime.MaxValue for the *DataEncoding*.

The representation of a *DateTime* on a *DevelopmentPlatform* also has a maximum precision. Decoders shall truncate *DateTime* values that exceed the supported precision. All *DevelopmentPlatforms* shall support a precision of at least 1ms.

The *DataValue* built-in type adds additional fields for Picoseconds. If a *DevelopmentPlatform* cannot support the full precision of *DateTime* values allowed by the *DataEncoding* then it should expand the size of its internal representation of Picoseconds field to preserve the full precision of the *DateTime*. If it does not do this it shall set the Picoseconds to 0.

The Picoseconds shall be set to 0 when the *DateTime* value is DateTime.MinValue or DateTime.MaxValue.

Concrete examples can be found in 5.2.2.5.

5.1.5. ByteString

A *ByteString* is structurally the same as a one-dimensional array of *Byte*. It is represented as a distinct built-in data type because it allows encoders to optimize the transmission of the value.

However, some *DevelopmentPlatforms* will not be able to preserve the distinction between a *ByteString* and a one-dimensional array of *Byte*.

If a decoder for *DevelopmentPlatform* cannot preserve the distinction it shall convert all one-dimensional arrays of *Byte* to *ByteStrings*.

Each element in a one-dimensional array of *ByteString* can have a different length which means is structurally different from a two-dimensional array of *Byte* where the length of each dimension is the same. This means decoders shall preserve the distinction between two or more dimension arrays of *Byte* and one or more dimension arrays of *ByteString*.

If a *DevelopmentPlatform* does not support unsigned integers, then it will have to represent *ByteStrings* as arrays of *SByte*. In this case, the requirements for *Byte* would then apply to *SByte*.

5.1.6. Number, Integer and UInteger

Number, *Integer* and *UInteger* are abstract simple types defined in IEC 62541-3. When these types are used in *Structure* fields, the field value is encoded as a *Variant*.

5.1.7. Structures and Unions

Structures are sequences of name value pairs defined by the *DataTypeDefinition Attribute* in IEC 62541-3. Each *DataEncoding* describes how to use the *DataTypeDefinition* to serialize *Structures*. *Subtypes* of *Structure* extend the parent by adding additional name-value pairs to the sequence.

If a *DataTypeDefinition* sets the *StructureType* to *StructureWithSubtypedValues* then any field with a subtype of *Structure DataType* and *IsOptional*=TRUE allows the type and any subtype of the field's *DataType* to be present in the field. In these cases, the values are serialized as an *ExtensionObject*. If *IsOptional*=FALSE then the field value is encoded directly according to the rules for the *DataEncoding*.

A field is serialized as an *ExtensionObject* if a field has a *DataType* set explicitly to *Structure*.

Unions are special subtypes of *Structure* where only one field value is encoded. All subtypes of *Union* are concrete. The remainder of the rules for *Structure* also apply to *Unions*.

5.1.8. ExtensionObject

An *ExtensionObject* is a container for any *Structure* and *Union DataTypes*.

The *ExtensionObject* contains a complex value serialized as a sequence of other *DataTypes*. It also contains an identifier which indicates what data it contains and how it is encoded.

There are four primary use cases where *ExtensionObjects* appear:

- When encoding a top-level *DataType* as a *Service Request* or *Response*;
- When encoding a *Structure* value inside a *Variant*;
- When encoding a field value in a *Structure* where the field *DataType* is *Structure*.
- When encoding a field value in a *Structure* where *AllowSubTypes*=TRUE (see F.12).

In all of these cases, the *ExtensionObject* provides an identifier that allows a decoder to know if it understands the *Structure* contained with it and a length that allows the *Structure* to be skipped if it is not recognized.

Structured DataTypes are represented in a *Server* address space as sub-types of the *Structure DataType*. The *DataEncodings* available for any given *Structured DataTypes* are represented as a *DataTypeEncoding Object* in the *Server AddressSpace*. The *NodeId* for the *DataTypeEncoding Object* is the identifier stored in the *ExtensionObject*. IEC 62541-3 describes how *DataTypeEncoding Nodes* are related to other *Nodes* of the *AddressSpace*.

Elements of an array of *ExtensionObjects* may have different *DataTypeEncoding NodeIds* specified. In some cases, this will be invalid, however, it is the responsibility of the application

layer to enforce whatever constraints are imposed by the Information Model on a given array. Decoders shall accept any valid *ExtensionObject* as an array element.

Server implementers should use namespace qualified numeric *Nodelds* for any *DataTypeEncoding Objects* they define. This will minimize the overhead introduced by packing *Structured DataType* values into an *ExtensionObject*.

ExtensionObjects and Variants allow unlimited nesting which could result in stack overflow errors even if the message size is less than the maximum allowed. Decoders shall support at least 100 nesting levels. Decoders shall report an error if the number of nesting levels exceeds what it supports.

5.1.9. Variant

A *Variant* is a union of all built-in data types including an *ExtensionObject*. *Variants* can also contain arrays of any of these built-in types. *Variants* are used to store any value or parameter with a data type of *BaseDataType* or one of its subtypes.

Variants can be empty. An empty *Variant* is described as having a null value and should be treated like a null column in a SQL database. A null value in a *Variant* may not be the same as a null value for data types that support nulls such as *Strings*. Some *DevelopmentPlatforms* may not be able to preserve the distinction between a null for a *DataType* and a null for a *Variant*, therefore, applications shall not rely on this distinction. This requirement also means that if an *Attribute* supports the writing of a null value it shall also support writing of an empty *Variant* and vice versa.

Variants can contain arrays of *Variants* but they cannot directly contain another *Variant*.

DiagnosticInfo types only have meaning when returned in a response message with an associated *StatusCode* and table of strings. As a result, *Variants* cannot contain instances of *DiagnosticInfo*.

Values of *Attributes* are always returned in instances of *DataValues*. Therefore, the *DataType* of an *Attribute* cannot be a *DataValue*. *Variants* can contain *DataValue* when used in other contexts such as *Method Arguments* or *PubSub Messages*. The *Variant* in a *DataValue* cannot, directly or indirectly, contain another *DataValue*.

ExtensionObjects and Variants allow unlimited nesting which could result in stack overflow errors even if the message size is less than the maximum allowed. Decoders shall support at least 100 nesting levels. Decoders shall report an error if the number of nesting levels exceeds what it supports.

5.1.10. Decimal

A *Decimal* is a high-precision signed decimal number. It consists of an arbitrary precision integer unscaled value and an integer scale. The scale is the inverse power of ten that is applied to the unscaled value.

A *Decimal* has the fields described in Table 3.

Table 3 – Layout of Decimal

Field	Type	Description
Typeld	Nodeld	The identifier for the <i>Decimal DataType</i> .
Encoding	Byte	This value is always 1.
Length	Int32	The length of the <i>Scale</i> and <i>Value</i> fields in bytes. If the length is less than or equal to 2 then the <i>Decimal</i> is an invalid value that cannot be used.
Scale	Int16	A signed integer representing scale which is the inverse power of ten that is applied to the unscaled value. i.e., the decimal number of the value multiplied by $10^{-\text{scale}}$ The integer is encoded starting with the least significant bit.
Value	OctetString	A 2-complement signed integer representing the unscaled value. The number of bytes is the value of the <i>Length</i> field minus size of the <i>Scale</i> field. The integer is encoded with the least significant byte first.

When a *Decimal* is encoded in a *Variant* the built-in type is set to *ExtensionObject*. Decoders that do not understand the *Decimal* type shall treat it like any other unknown *Structure* and pass it on to the application. Decoders that do understand the *Decimal* can parse the value and use any construct that is suitable for the *DevelopmentPlatform*. Note that a *Decimal* is like a built-in type and a *DevelopmentPlatform* has to have hardcoded knowledge of the type. No *Structure* metadata is published for this type.

If a *Decimal* is embedded in another *Structure* then the *DataTypeDefinition* for the field shall specify the *NodeId* of the *Decimal* Node as the *DataType*. If a *Server* publishes an OPC Binary type description for the *Structure* then the type description shall set the *DataType* for the field to *ExtensionObject*.

5.1.11. Null, Empty and Zero-Length Arrays

The terms null, empty and zero-length are used to describe array values (*Strings* are arrays of characters and *ByteStrings* are arrays of *Bytes* for purposes of this discussion). A null array has no value. A zero-length or empty array is an array with 0 elements. Some *DataEncodings* will allow the distinction to be preserved on the wire, however, not all *DevelopmentPlatforms* will be able to preserve the distinction. For this reason, null, empty and zero length arrays are semantically the same for all *DataEncodings*. Decoders shall be able to handle all variations supported by the *DataEncoding*, however, decoders are not required to preserve the distinction. When testing for equality, applications shall treat null and empty arrays as equal. When a *DevelopmentPlatform* supports the distinction, writing and reading back an array value may result in null array becoming an empty array or vice versa.

5.1.12. QualifiedName, NodeId and ExpandedNodeId String Encoding

QualifiedNames, *NodeIds* and *ExpandedNodeIds* optimize representation of *NamespaceUris* by using a *NamespaceIndex* to reference the location of the URI in a *NamespaceTable*. The *NamespaceTable* to use is known from the context. For example, when a *Client* establishes a *Session* with a *Server*, the *Server* supplies the *NamespaceTable* that is used for all exchanges within that *Session*. Another example, is when a *Publisher* publishes *DataSetMetadata* messages, the *NamespaceTable* is provided as part of the message.

However, there are other scenarios where there is no obvious context that can be used to store the *NamespaceTable*, such as column in a database table, so it is necessary to provide a self-contained representation of these *DataTypes*. This clause defines a normative *String* representation using the ABNF notation (see RFC 5234). Table 4 defines additional core rules used in these definitions.

Table 4 – Additional ABNF Core Rules

UNICODE	Any Unicode character other than a Control character.
CONTROL	Any Unicode Control character (includes nulls, carriage returns, tabs and new lines).
URI	A string that conforms to RFC 3986.
ENCODEDURI	A URI which has the RFC 3986 Percent-Encoding applied to it.
BASE64	A Base64 encoded binary value (see Base64).

The ABNF description for a *NodeId* is found in Table 5.

Table 5 – ABNF Description for a NodeId

<node-id>	= <identifier>
<node-id>	= / <namespace-index> ";" <identifier>
<node-id>	= / <namespace-uri> ";" <identifier>
<namespace-index>	= "ns=" *DIGIT
<namespace-uri>	= "nsu=" ENCODEDURI
<identifier>	= <numericid> / <stringid> / <guidid> / <opaqueid>
<numericid>	= "i=" *DIGIT
<stringid>	= "s=" *(UNICODE)

<guidid>	= "g=" 8HEXDIG 3("-" 4HEXDIG) "-" 12HEXDIG
<opaqueid>	= "b=" BASE64
<reserved>	= ";" / "%" / CONTROL
<escaped>	= "%" 2HEXDIG

NodeIds with a *NamespaceIndex* of 0 or a *NamespaceUri* of `http://opcfoundation.org/UA/` shall only use the <identifier> form.

The <escaped> form of a Unicode character is the UTF-8 encoding of the Unicode character represented as a "%" followed by a hexadecimal number. Examples of *NodeIds*:

```
i=13
ns=10;i=12345
nsu=http://widgets.com/schemas/hello;s=水%09World
g=09087e75-8e5e-499b-954f-f2a9603db28a
nsu=tag:acme.com,2023:schemas:data#off%3B;b=M/RbKBsRVkePCePcx24oRA==
```

The ABNF description for a *ExpandedNodeId* is found in Table 6.

Table 6 – ABNF Description for a ExpandedNodeId

<expanded-node-id>	= <node-id>
<expanded-node-id>	=/ <server-index> ";" <node-id>
<expanded-node-id>	=/ <server-uri> ";" <node-id>
<server-index>	= "svr=" *DIGIT
<server-uri>	= "svu=" ENCODEDURI

ExpandedNodeIds that are not specific to a *Server* shall use the <node-id> form.

Examples of *ExpandedNodeIds*:

```
i=13
svr=1;nsu=http://widgets.com/schemas/hello;s=水%09World
svu=http://smith.com/east/factory;g=09087e75-8e5e-499b-954f-f2a9603db28a
svu=http://smith.com/west/factory;nsu=tag:acme.com,2023:schemas:data#off%3B;b=M/Rb
KBsRVkePCePcx24oRA==
```

The ABNF description for a *QualifiedName* is found in Table 7.

Table 7 – ABNF Description for a QualifiedName

<qualified-name>	= <name>
<qualified-name>	=/ <namespace-index> ":" <name>
<qualified-name>	=/ <namespace-uri> ";" <name>
<name>	= *(UNICODE)

QualifiedNames in the OPC UA Namespace shall not use the forms with a <namespace-index> or a <namespace-uri>.

Examples of *QualifiedNames*:

```
InputArguments
3:Hello%09World
nsu=http://widgets.com/schemas/hello;Hello%3BWorld
nsu=tag:acme.com,2023:schemas:data#off%3B;Boiler2
```

5.1.13. Name Encoding Rules

Text based *DataEncodings* such as UA XML (5.3) or UA JSON (5.4) make use of *Names* from the *DataType* and *DataTypeDefinition* to create the serialized data. XML and JSON formats impose restrictions on the *Strings* that may be used in serialized data so it is necessary to define a transformation that ensures any *Names* can be used in the *DataEncoding*. The following rules are used to convert a *DataType* Name or a Structure Field Name to a string supported by the Encoding:

- 1) Any character that is not permitted by the *DataEncoding* is replaced by an underscore (U+005F);
- 2) A name with text sequence that is not valid in the first position has an underscore (U+005F) added as a prefix.

The character restrictions for the XML *DataEncoding* are:

<allowed-name>	= <letter> *(<allowed-char> / "_" / "-" / ".")
<allowed-char>	= <letter> / DIGIT
<letter>	= UNICODE-LETTER
UNICODE-LETTER	Any Unicode character with a general category of 'Letter'

In addition, XML names cannot start with the text 'xml' or any variation in case (i.e. 'xMI').

There are no restrictions for the JSON encoding other than the general restrictions defined in IEC 62541-3.

Table 8 has examples of XML encoded names.

Table 8 – Examples of XML Encoded Names

Hello	Hello
_Hello	_Hello
3DHello	_3DHello
冷水	冷水
3 (冷水。)-Hello	_3__冷水____-Hello

5.2. OPC UA Binary

5.2.1. General

The OPC UA *Binary DataEncoding* is a data format developed to meet the performance needs of OPC UA applications. This format is designed primarily for fast encoding and decoding, however, the size of the encoded data on the wire was also a consideration.

The OPC UA *Binary DataEncoding* relies on several primitive data types with clearly defined encoding rules that can be sequentially written to or read from a binary stream. A structure is encoded by sequentially writing the encoded form of each field. If a given field is also a structure, then the values of its fields are written sequentially before writing the next field in the containing structure. All fields shall be written to the stream even if they contain null values. The encodings for each primitive type specify how to encode either a null or a *DefaultValue* for the type.

The OPC UA *Binary DataEncoding* does not include any type or field name information because all OPC UA applications are expected to have advance knowledge of the services and structures that they support. An exception is an *ExtensionObject* which provides an identifier and a size for the *Structured DataType* structure it represents. This allows a decoder to skip over types that it does not recognize.

5.2.2. Built-in Types

5.2.2.1. Boolean

A *Boolean* value shall be encoded as a single byte where a value of 0 (zero) is false and any non-zero value is true.

Encoders shall use the value of 1 to indicate a true value; however, decoders shall treat any non-zero value as true.

5.2.2.2. Integer

All integer types shall be encoded as little-endian values where the least significant byte appears first in the stream.

Figure 2 illustrates how value 1 000 000 000 (Hex: 3B9ACA00) is encoded as a 32-bit integer in the stream.

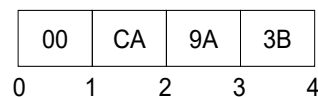


Figure 2 – Encoding Integers in a binary stream

5.2.2.3. Floating Point

All floating-point values shall be encoded with the appropriate IEEE 754 binary representation which has three basic components: the sign, the exponent, and the fraction. The bit ranges assigned to each component depend on the width of the type. Table 9 lists the bit ranges for the supported floating-point types.

Table 9 – Supported Floating Point Types

Name	Width (bits)	Fraction	Exponent	Sign
Float	32	0-22	23-30	31
Double	64	0-51	52-62	63

In addition, the order of bytes in the stream is significant. All floating-point values shall be encoded with the least significant byte appearing first (i.e., little endian).

Figure 3 illustrates how the value -6,5 (Hex: C0D00000) is encoded as a *Float*.

The floating-point type supports positive and negative infinity and not-a-number (NaN). The IEEE specification allows for multiple NaN variants; however, the encoders/decoders may not preserve the distinction. Encoders shall encode a NaN value as an IEEE quiet-NAN (Hex: 000000000000F8FF) or (Hex: 0000C0FF). Any unsupported types such as denormalized numbers shall also be encoded as an IEEE quiet-NAN. Any test for equality between NaN values always fails.

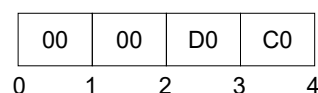


Figure 3 – Encoding Floating Points in a binary stream

5.2.2.4. String

All *String* values are encoded as a sequence of UTF-8 characters preceded by the length in bytes.

The length in bytes is encoded as *Int32*. A value of -1 is used to indicate a 'null' string.

Strings with embedded nulls ('\0') can lead to unpredictable application behaviour because embedded nulls have special meaning to some *DevelopmentPlatforms*. For this reason, embedded nulls are not recommended and *ByteString* should be used instead. That said, Encoders may encode *Strings* with embedded nulls. Decoders shall use the length to read all bytes in *String*.

Figure 4 illustrates how the multilingual string '水Boy' is encoded in a byte stream.

Length				水			B	o	y	
06	00	00	00	E6	B0	B4	42	6F	79	
0	1	2	3	4	5	6	7	8	9	10

Figure 4 – Encoding Strings in a binary stream

5.2.2.5. DateTime

A *DateTime* value shall be encoded as a 64-bit signed integer (see 5.2.2.2) which represents the number of 100 nanosecond intervals since January 1, 1601 (UTC).

Not all *DevelopmentPlatforms* will be able to represent the full range of dates and times that can be represented with this *DataEncoding*. For example, the UNIX *time_t* structure only has a 1 second resolution and cannot represent dates prior to 1970. For this reason, a number of rules shall be applied when dealing with date/time values that exceed the dynamic range of a *DevelopmentPlatform*. These rules are:

- n) A date/time value is encoded as 0 if either
 - 1) The value is equal to or earlier than 1601-01-01 12:00AM UTC.
 - 2) The value is the earliest date that can be represented with the *DevelopmentPlatform's* encoding.
- o) A date/time is encoded as the maximum value for an *Int64* if either
 - 3) The value is equal to or greater than 9999-12-31 11:59:59PM UTC,
 - 4) The value is the latest date that can be represented with the *DevelopmentPlatform's* encoding.
- p) A date/time is decoded as the earliest time that can be represented on the platform if either
 - 5) The encoded value is 0,
 - 6) The encoded value represents a time earlier than the earliest time that can be represented with the *DevelopmentPlatform's* encoding.
- q) A date/time is decoded as the latest time that can be represented on the platform if either
 - 7) The encoded value is the maximum value for an *Int64*,
 - 8) The encoded value represents a time later than the latest time that can be represented with the *DevelopmentPlatform's* encoding.

These rules imply that the earliest and latest times that can be represented on a given platform are invalid date/time values and should be treated that way by applications.

A decoder shall truncate the value if a decoder encounters a *DateTime* value with a resolution that is greater than the resolution supported on the *DevelopmentPlatform*.

5.2.2.6. Guid

A *Guid* is encoded in a structure as shown in Table 2. Fields are encoded sequentially according to the data type for field.

Figure 5 illustrates how the *Guid* "72962B91-FA75-4AE6-8D28-B404DC7DAF63" is encoded in a byte stream.

Data1				Data2		Data3		Data4								
91	2B	96	72	75	FA	E6	4A	8D	28	B4	04	DC	7D	AF	63	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 5 – Encoding Guids in a binary stream

5.2.2.7. ByteString

A *ByteString* is encoded as sequence of bytes preceded by its length in bytes. The length is encoded as a 32-bit signed integer as described above.

If the length of the byte string is –1 then the byte string is 'null'.

5.2.2.8. XmlElement

An *XmlElement* is an XML element serialized as UTF-8 string and then encoded as *ByteString*.

Figure 6 illustrates how the *XmlElement* "<A>Hot 水" is encoded in a byte stream.

Length				<A>			Hot			水							
0D	00	00	00	3C	41	3E	48	6F	74	E6	B0	B4	3C	2F	41	3E	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Figure 6 – Encoding XmlElement in a binary stream

A decoder may choose to parse the XML after decoding; if an unrecoverable parsing error occurs then the decoder should try to continue processing the stream. For example, if the *XmlElement* is the body of a *Variant* or an element in an array which is the body of a *Variant* then this error can be reported by setting value of the *Variant* to the *StatusCode Bad_DecodingError*.

5.2.2.9. NodeId

The components of a *NodeId* are described the Table 10.

Table 10 – NodeId components

Name	Data Type	Description
Namespace	UInt16	The index for a namespace URI. An index of 0 is used for OPC UA defined <i>NodeIds</i> .
IdentifierType	Enumeration	The format and data type of the identifier. The value may be one of the following: NUMERIC - the value is an <i>UInteger</i> ; STRING - the value is <i>String</i> ; GUID - the value is a <i>Guid</i> ; OPAQUE - the value is a <i>ByteString</i> ;
Value	UInt32 or String or Guid or ByteString	The identifier for a node in the address space of an OPC UA <i>Server</i> .

The *DataEncoding* of a *NodeId* varies according to the contents of the instance. For that reason, the first byte of the encoded form indicates the format of the rest of the encoded *NodeId*. The possible *DataEncoding* formats are shown in Table 11. Table 11 through Table 14 describe the structure of each possible format (they exclude the byte which indicates the format).

Table 11 – NodeId DataEncoding values

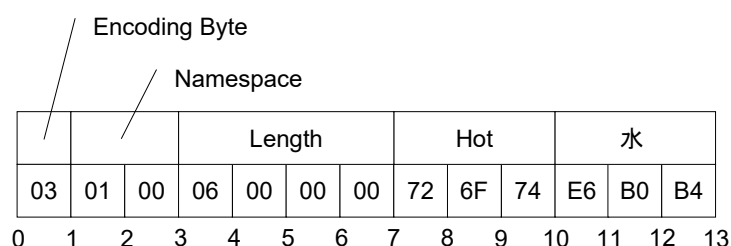
Name	Value	Description
Two Byte	0x00	A numeric value that fits into the two-byte representation.
Four Byte	0x01	A numeric value that fits into the four-byte representation.
Numeric	0x02	A numeric value that does not fit into the two or four byte representations.
String	0x03	A String value.
Guid	0x04	A Guid value.
ByteString	0x05	An opaque (ByteString) value.
NamespaceUri Flag	0x80	See discussion of <i>ExpandedNodeId</i> in 5.2.2.10.
ServerIndex Flag	0x40	See discussion of <i>ExpandedNodeId</i> in 5.2.2.10.

The standard *NodeId DataEncoding* has the structure shown in Table 12. The standard *DataEncoding* is used for all formats that do not have an explicit format defined.

Table 12 – Standard NodeId Binary DataEncoding

Name	Data Type	Description
Namespace	UInt16	The <i>NamespaceIndex</i> .
Identifier	*	The identifier which is encoded according to the following rules: NUMERIC UInt32 STRING String GUID Guid OPAQUE ByteString

An example of a String *NodeId* with Namespace = 1 and Identifier = “Hot 水” is shown in Figure 7.

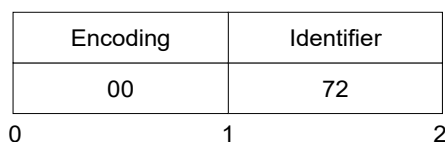
**Figure 7 – A String NodeId**

The Two Byte *NodeId DataEncoding* has the structure shown in Table 13.

Table 13 – Two Byte NodeId Binary DataEncoding

Name	Data Type	Description
Identifier	Byte	The <i>Namespace</i> is the default OPC UA namespace (i.e. 0). The <i>Identifier</i> Type is 'Numeric'. The <i>Identifier</i> shall be in the range 0 to 255.

An example of a Two Byte *NodeId* with Identifier = 72 is shown in Figure 8.

**Figure 8 – A Two Byte NodeId**

The Four Byte *NodeId* *DataEncoding* has the structure shown in Table 14.

Table 14 – Four Byte *NodeId* Binary *DataEncoding*

Name	Data Type	Description
Namespace	Byte	The <i>Namespace</i> shall be in the range 0 to 255.
Identifier	UInt16	The <i>Identifier</i> Type is 'Numeric'. The <i>Identifier</i> shall be an integer in the range 0 to 65 535.

An example of a Four Byte *NodeId* with Namespace = 5 and Identifier = 1 025 is shown in Figure 9.

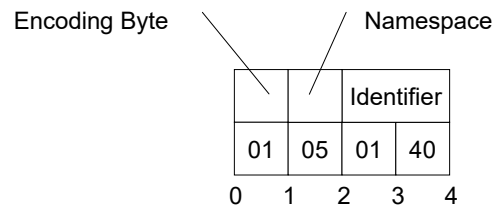


Figure 9 – A Four Byte *NodeId*

5.2.2.10. *ExpandedNodeId*

An *ExpandedNodeId* extends the *NodeId* structure by allowing the *NamespaceUri* to be explicitly specified instead of using the *NamespaceIndex*. The *NamespaceUri* is optional. If it is specified, then the *NamespaceIndex* inside the *NodeId* shall be ignored.

The *ExpandedNodeId* is encoded by first encoding a *NodeId* as described in 5.2.2.9 and then encoding *NamespaceUri* as a *String*.

An instance of an *ExpandedNodeId* may still use the *NamespaceIndex* instead of the *NamespaceUri*. In this case, the *NamespaceUri* is not encoded in the stream. The presence of the *NamespaceUri* in the stream is indicated by setting the *NamespaceUri* flag in the encoding format byte for the *NodeId*.

If the *NamespaceUri* is present, then the encoder shall encode the *NamespaceIndex* as 0 in the stream when the *NodeId* portion is encoded. The unused *NamespaceIndex* is included in the stream for consistency.

An *ExpandedNodeId* may also have a *ServerIndex* which is encoded as a *UInt32* after the *NamespaceUri*. The *ServerIndex* flag in the *NodeId* encoding byte indicates whether the *ServerIndex* is present in the stream. The *ServerIndex* is omitted if it is equal to zero.

The *ExpandedNodeId* encoding has the structure shown in Table 15.

Table 15 – *ExpandedNodeId* Binary *DataEncoding*

Name	Data Type	Description
<i>NodeId</i>	<i>NodeId</i>	The <i>NamespaceUri</i> and <i>ServerIndex</i> flags in the <i>NodeId</i> encoding indicate whether those fields are present in the stream.
<i>NamespaceUri</i>	String	Not present if null or Empty.
<i>ServerIndex</i>	UInt32	Not present if 0.

5.2.2.11. *StatusCode*

A *StatusCode* is encoded as a *UInt32*.

5.2.2.12. DiagnosticInfo

A *DiagnosticInfo* structure is described in IEC 62541-4. It specifies a number of fields that could be missing. For that reason, the encoding uses a bit mask to indicate which fields are actually present in the encoded form.

As described in IEC 62541-4, the *SymbolicId*, *NamespaceUri*, *LocalizedText* and *Locale* fields are indexes in a string table which is returned in the response header. Only the index of the corresponding string in the string table is encoded. An index of -1 indicates that there is no value for the string.

DiagnosticInfo allows unlimited nesting which could result in stack overflow errors even if the message size is less than the maximum allowed. Decoders shall support at least 100 nesting levels. Decoders shall report an error if the number of nesting levels exceeds what it supports.

Table 16 – DiagnosticInfo Binary DataEncoding

Name	Data Type	Description
Encoding Mask	Byte	A bit mask that indicates which fields are present in the stream. The mask has the following bits: 0x01 Symbolic Id 0x02 Namespace 0x04 LocalizedText 0x08 Locale 0x10 Additional Info 0x20 InnerStatusCode 0x40 InnerDiagnosticInfo
SymbolicId	Int32	A symbolic name for the status code.
NamespaceUri	Int32	A namespace that qualifies the symbolic id.
Locale	Int32	The locale used for the localized text.
LocalizedText	Int32	A human readable summary of the status code.
Additional Info	String	Detailed application specific diagnostic information.
Inner StatusCode	StatusCode	A status code provided by an underlying system.
Inner DiagnosticInfo	DiagnosticInfo	Diagnostic info associated with the inner status code.

5.2.2.13. QualifiedName

A *QualifiedName* structure is encoded as shown in Table 17.

The abstract *QualifiedName* structure is defined in IEC 62541-3.

Table 17 – QualifiedName Binary DataEncoding

Name	Data Type	Description
NamespaceIndex	UInt16	The namespace index.
Name	String	The name.

5.2.2.14. LocalizedText

A *LocalizedText* structure contains two fields that could be missing. For that reason, the encoding uses a bit mask to indicate which fields are actually present in the encoded form.

The abstract *LocalizedText* structure is defined in IEC 62541-3.

Table 18 – LocalizedText Binary DataEncoding

Name	Data Type	Description
EncodingMask	Byte	A bit mask that indicates which fields are present in the stream. The mask has the following bits: 0x01 Locale 0x02 Text
Locale	String	The locale. Omitted is null or empty.
Text	String	The text in the specified locale. Omitted is null or empty.

5.2.2.15.ExtensionObject

An *ExtensionObject* is encoded as sequence of bytes prefixed by the *NodeId* of its *DataTypeEncoding*, the *DataEncoding* used and the number of bytes encoded.

An *ExtensionObject* may be serialized as a *ByteString* or an *XmlElement* by the application and then passed to the encoder. In this case, the encoder will be able to write the number of bytes in the object before it encodes the bytes. However, an *ExtensionObject* may know how to encode/decode itself which means the encoder shall calculate the number of bytes before it encodes the object or it shall be able to seek backwards in the stream and update the length after encoding the body.

When a decoder encounters an *ExtensionObject* it shall check if it recognizes the *DataTypeEncoding* identifier. If it does, then it can call the appropriate function to decode the object body. If the decoder does not recognize the type it shall use the *Encoding* to determine if the body is a *ByteString* or an *XmlElement* and then decode the object body or treat it as opaque data and skip over it.

The serialized form of an *ExtensionObject* is shown in Table 19.

Table 19 – Extension Object Binary DataEncoding

Name	Data Type	Description
TypeId	NodeId	The identifier for the <i>DataTypeEncoding</i> node in the <i>Server's AddressSpace</i> . <i>ExtensionObjects</i> defined by the OPC UA specification have a numeric node identifier assigned to them with a <i>NamespaceIndex</i> of 0. The numeric identifiers are defined in A.3. Decoders use this field to determine the syntax of the <i>Body</i> . For example, if this field is the <i>NodeId</i> of the <i>JSON Encoding Object</i> for a <i>DataType</i> then the <i>Body</i> is a <i>ByteString</i> containing a JSON document encoded as a UTF-8 string.
Encoding	Byte	An enumeration that indicates how the body is encoded. The parameter may have the following values: 0x00 No body is encoded. 0x01 The body is encoded as a <i>ByteString</i> . 0x02 The body is encoded as an <i>XmlElement</i> .
Length	Int32	The length of the object body. The length shall be specified if the body is encoded.
Body	OctetString	The encoded object. This field contains the raw bytes for <i>ByteString</i> bodies. For <i>XmlElement</i> bodies this field contains the XML encoded as a UTF-8 string without any null terminator. Some binary encoded structures may have a serialized length that is not a multiple of 8 bits. Encoders shall append 0 bits to ensure the serialized length is a multiple of 8 bits. Decoders that understand the serialized format shall ignore the padding bits.

A decoder may choose to parse an *XmlElement* body after decoding; if an unrecoverable parsing error occurs then the decoder should try to continue processing the stream. For example, if the *ExtensionObject* is the body of a *Variant* or an element in an array that is the body of

1025 *Variant* then this error can be reported by setting the value of the *Variant* to the *StatusCode*
 1026 *Bad_DecodingError*.

1027 5.2.2.16.Variant

1028 A *Variant* is a union of the built-in types.

1029 The structure of a *Variant* is shown in Table 20.

1030 **Table 20 – Variant Binary DataEncoding**

Name	Data Type	Description
EncodingMask	Byte	The type of data encoded in the stream. A value of 0 specifies a NULL and that no other fields are encoded. The mask has the following bits assigned: 0:5 Built-in Type Id (see Table 1). 6 True if the ArrayDimensions field is encoded. 7 True if an array of values is encoded. The Built-in Type Ids 26 through 31 are not currently assigned but may be used in the future. Decoders shall accept these IDs, assume the <i>Value</i> contains a <i>ByteString</i> or an array of <i>ByteStrings</i> and pass both onto the application. Encoders shall not use these IDs. The <i>ArrayDimensions</i> field shall only be present if the number of dimensions is 2 or greater and all dimensions have a length greater than 0.
ArrayLength	Int32	The number of elements in the array. This field is only present if the array bit is set in the encoding mask. Multi-dimensional arrays are encoded as a one-dimensional array and this field specifies the total number of elements. The original array can be reconstructed from the dimensions that are encoded after the value field. Higher rank dimensions are serialized first. For example, an array with dimensions [2,2,2] is written in this order: [0,0,0], [0,0,1], [0,1,0], [0,1,1], [1,0,0], [1,0,1], [1,1,0], [1,1,1] If one or more dimensions has a length <= 0 then the ArrayLength is 0.
Value	*	The value encoded according to its built-in data type. If the array bit is set in the encoding mask, then each element in the array is encoded sequentially. Since many types have variable length encoding each element shall be decoded in order. The value shall not be a <i>Variant</i> but it could be an array of <i>Variants</i>. Many implementation platforms do not distinguish between one dimensional Arrays of <i>Bytes</i> and <i>ByteStrings</i>. For this reason, decoders are allowed to automatically convert an Array of <i>Bytes</i> to a <i>ByteString</i>.
ArrayDimensions Length	Int32	The number of dimensions. This field is only present if the ArrayDimensions flag is set in the encoding mask.
ArrayDimensions	Int32[]	The length of each dimension encoded as a sequence of Int32 values This field is only present if the ArrayDimensions flag is set in the encoding mask. The lower rank dimensions appear first in the array. All dimensions shall be specified and shall be greater than zero.. If <i>ArrayDimensions</i> are inconsistent with the <i>ArrayLength</i> then the decoder shall stop and raise a <i>Bad_DecodingError</i>.

1031
 1032 The types and their identifiers that can be encoded in a *Variant* are shown in Table 1.

1033 5.2.2.17.DataValue

1034 A *DataValue* is always preceded by a mask that indicates which fields are present in the stream.

1035 The fields of a *DataValue* are described in Table 21.

1036

Table 21 – Data Value Binary DataEncoding

Name	Data Type	Description
Encoding Mask	Byte	A bit mask that indicates which fields are present in the stream. The mask has the following bits: 0x01 False if the Value is <i>Null</i> . 0x02 False if the StatusCode is Good. 0x04 False if the SourceTimestamp is <i>DateTime.MinValue</i> . 0x08 False if the ServerTimestamp is <i>DateTime.MinValue</i> . 0x10 False if the SourcePicoSeconds is not present. 0x20 False if the ServerPicoSeconds is not present.
Value	Variant	The value. Not present if the Value bit in the EncodingMask is False.
Status	StatusCode	The status associated with the value. Not present if the StatusCode bit in the EncodingMask is False.
SourceTimestamp	DateTime	The source timestamp associated with the value. Not present if the SourceTimestamp bit in the EncodingMask is False.
SourcePicoSeconds	UInt16	The number of 10 Picosecond intervals for the SourceTimestamp. Not present if the SourcePicoSeconds bit in the EncodingMask is False. If the source timestamp is missing the PicoSeconds are ignored.
ServerTimestamp	DateTime	The Server timestamp associated with the value. Not present if the ServerTimestamp bit in the EncodingMask is False.
ServerPicoSeconds	UInt16	The number of 10 Picosecond intervals for the ServerTimestamp. Not present if the ServerPicoSeconds bit in the EncodingMask is False. If the Server timestamp is missing the PicoSeconds are ignored.

1037

1038 The *PicoSeconds* fields store the difference between a high-resolution timestamp with a
 1039 resolution of 10 PicoSeconds and the *Timestamp* field value which only has a 100 ns resolution.
 1040 The *PicoSeconds* fields shall contain values less than 10 000. The decoder shall treat values
 1041 greater than or equal to 10 000 as the value '9999'.

1042 5.2.3. Decimal

1043 *Decimals* are encoded as described in 5.1.10.

1044 A *Decimal* does not have a NULL value.

1045 5.2.4. Enumerations

1046 Enumerations are encoded as *Int32* values.

1047 An *Enumeration* does not have a NULL value.

1048 5.2.5. Arrays

1049 One dimensional *Arrays* are encoded as a sequence of elements preceded by the number of
 1050 elements encoded as an *Int32* value.

1051 Multi-dimensional *Arrays* have an encoding that depends on where they are used. When a multi-
 1052 dimensional *Array* is the *Value* of an *Attribute* it uses the *Variant* encoding described in 5.2.2.16.

1053 When a multi-dimensional *Array* is a field of a *Structure* (see 5.2.6) it shall be encoded with the
 1054 inline matrix representation as shown in Table 22.

Table 22 – Inline Matrix DataEncoding

Name	Data Type	Description
Dimensions	Int32 []	The length of each dimension. If any dimension has a length <= 0, then no values are encoded. The number of dimensions shall be at least 2.
Values	*	The values encoded sequentially according to its built-in data type. The total number of values is the product of the dimensions. The mapping of a multidimensional array to a flat list is described in 5.2.2.16.

The inline matrix representation is not supported by earlier versions of this specification. This means any *Structure* with a field or nested field mapped to an inline matrix is not compatible with the deprecated *DataTypeDictionary* mechanism, and shall not be included in a *DataTypeDictionary*.

If an *Array* is null, then its length is encoded as -1. See 5.1.11 for a discussion of zero-length vs null arrays.

5.2.6. Structures

Structures are encoded as a sequence of fields in the order that they appear in the definition. The encoding for each field is determined by the built-in type for the field.

All fields specified in the structure shall be encoded. If optional fields exist in the *Structure*, then see 5.2.7.

Structures do not have a null value. If an encoder is written in a programming language that allows structures to have null values, then the encoder shall create a new instance with *DefaultValues* for all fields and serialize that. Encoders shall not generate an encoding error in this situation.

The following is an example of a structure using C/C++ syntax:

```

struct Type2
{
    Int32 A;
    Int32 B;
};

struct Type1
{
    Int32 X;
    Byte NoOfY;
    Type2* Y;
    Int32 Z;
    UInt16 W[10];
    Byte M[2,3,4];
};

```

In the C/C++ example above, the Y field is a pointer to an array with a length stored in NoOfY. When encoding an array, the length is part of the array encoding so the NoOfY field is not encoded. That said, encoders and decoders use NoOfY during encoding. W is a fixed length array with an implicitly defined length of 10. This length is always encoded with the array. M is a fixed length multidimensional array. The length of each dimension is always encoded with the array.

An instance of *Type1* which contains an array of two *Type2* instances would be encoded as 28-byte sequence. If the instance of *Type1* was encoded in an *ExtensionObject* it would have an additional prefix shown in Table 23 which would make the total length 101 bytes. The *TypeId*, *Encoding* and the *Length* are fields defined by the *ExtensionObject*. The encoding of the *Type2* instances do not include any type identifier because it is explicitly defined in *Type1*.

1101

Table 23 – Sample OPC UA Binary Encoded structure

Field	Bytes	Value
Type Id	4	The identifier for the 'Type1' Binary Encoding Node
Encoding	1	0x1 for ByteString
Length	4	28
X	4	The value of field 'X'
Y.Length	4	2
Y.A	4	The value of field 'Y[0].A'
Y.B	4	The value of field 'Y[0].B'
Y.A	4	The value of field 'Y[1].A'
Y.B	4	The value of field 'Y[1].B'
Z	4	The value of field 'Z'
W.Length	4	10
W	20	The value of field 'W'.
M.Dimensions.Length	4	3
M.Dimensions	12	The sequence [2,3,4] encoded as a <i>Int32</i> .
M.Values	24	The values in 'M' encoded sequentially as described in 5.2.2.16.

1102

1103 The *Value* of the *DataTypeDefinition Attribute* for a *DataType Node* describing Type1 is shown
 1104 in Table 24.

1105 **Table 24 – DataTypeDefinition for “Type1” from Sample**

Name	Type	Description
defaultEncodingId	NodeId	NodeId of the “Type1_Encoding_DefaultBinary” Node.
baseDataType	NodeId	“i=22” [Structure]
structureType	StructureType	Structure_0 [Structure without optional fields]
fields [0]	StructureField	
name	String	“X”
description	LocalizedText	Description of X
dataType	NodeId	“i=6” [Int32]
valueRank	Int32	-1 (Scalar)
arrayDimensions	UInt32[]	null
maxStringLength	UInt32	0
isOptional	Boolean	false
fields [1]	StructureField	
name	String	“Y”
description	LocalizedText	Description of Y-Array
dataType	NodeId	NodeId of the Type2 DataType Node (e.g. “ns=3; s=MyType2”)
valueRank	Int32	1 (OneDimension)
arrayDimensions	UInt32[]	{ 0 }
maxStringLength	UInt32	0
isOptional	Boolean	false
fields [2]	StructureField	
name	String	“Z”
description	LocalizedText	Description of Z
dataType	NodeId	“i=6” [Int32]
valueRank	Int32	-1 (Scalar)
arrayDimensions	UInt32[]	null
maxStringLength	UInt32	0
isOptional	Boolean	false
fields [3]	StructureField	
name	String	“W”
description	LocalizedText	Description of W
dataType	NodeId	“i=5” [UInt16]
valueRank	Int32	1 (OneDimension)
arrayDimensions	UInt32[]	{ 10 }
maxStringLength	UInt32	0
isOptional	Boolean	false
fields [4]	StructureField	
name	String	“M”
description	LocalizedText	Description of M
dataType	NodeId	“i=3” [Byte]
valueRank	Int32	3
arrayDimensions	UInt32[]	{ 2, 3, 4 }
maxStringLength	UInt32	0
isOptional	Boolean	false

1106