



BSI Standards Publication

OPC Unified Architecture

Part 6: Mappings

National foreword

This British Standard is the UK implementation of EN IEC 62541-6:2020. It is identical to IEC 62541-6:2020. It supersedes [BS EN 62541-6:2015](#), which will be withdrawn on 17 August 2023.

The UK participation in its preparation was entrusted to Technical Committee GEL/65/3, Industrial communications: process measurement and control, including fieldbus.

A list of organizations represented on this committee can be obtained on request to its committee manager.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2020
Published by BSI Standards Limited 2020

ISBN 978 0 580 51135 6

ICS 25.040.40; 35.100.05; 25.100.01

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 30 September 2020.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------

English Version

OPC Unified Architecture - Part 6: Mappings (IEC 62541-6:2020)

Architecture unifiée OPC - Partie 6: Mappings
(IEC 62541-6:2020)

OPC Unified Architecture - Teil 6: Protokollabbildungen
(IEC 62541-6:2020)

This European Standard was approved by CENELEC on 2020-08-17. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

Up-to-date lists and bibliographical references concerning such national standards may be obtained on application to the CEN-CENELEC Management Centre or to any CENELEC member.

This European Standard exists in three official versions (English, French, German). A version in any other language made by translation under the responsibility of a CENELEC member into its own language and notified to the CEN-CENELEC Management Centre has the same status as the official versions.

CENELEC members are the national electrotechnical committees of Austria, Belgium, Bulgaria, Croatia, Cyprus, the Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, the Netherlands, Norway, Poland, Portugal, Republic of North Macedonia, Romania, Serbia, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey and the United Kingdom.



European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique
Europäisches Komitee für Elektrotechnische Normung

CEN-CENELEC Management Centre: Rue de la Science 23, B-1040 Brussels

European foreword

The text of document 65E/718/FDIS, future edition 3 of IEC 62541-6, prepared by SC 65E "Devices and integration in enterprise systems" of IEC/TC 65 "Industrial-process measurement, control and automation" was submitted to the IEC-CENELEC parallel vote and approved by CENELEC as EN IEC 62541-6:2020.

The following dates are fixed:

- latest date by which the document has to be implemented at national (dop) 2021-05-17 level by publication of an identical national standard or by endorsement
- latest date by which the national standards conflicting with the (dow) 2023-08-17 document have to be withdrawn

This document supersedes EN 62541-6:2015 and all of its amendments and corrigenda (if any).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. CENELEC shall not be held responsible for identifying any or all such patent rights.

This document has been prepared under a mandate given to CENELEC by the European Commission and the European Free Trade Association.

Endorsement notice

The text of the International Standard IEC 62541-6:2020 was approved by CENELEC as a European Standard without any modification.

Annex ZA

(normative)

Normative references to international publications with their corresponding European publications

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

NOTE 1 Where an International Publication has been modified by common modifications, indicated by (mod), the relevant EN/HD applies.

NOTE 2 Up-to-date information on the latest versions of the European Standards listed in this annex is available here: www.cenelec.eu.

<u>Publication</u>	<u>Year</u>	<u>Title</u>	<u>EN/HD</u>	<u>Year</u>
IEC/TR 62541-1	-	OPC unified architecture - Part 1: Overview and concepts	CLC/TR 62541-1	-
IEC/TR 62541-2	-	OPC unified architecture - Part 2: Security model	CLC/TR 62541-2	-
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 62541-7	-	OPC unified architecture - Part 7: Profiles	EN IEC 62541-7	-
IEC 62541-12	-	OPC unified architecture - Part 12: Discovery and global services	EN IEC 62541-12	-
ISO 8601-1	2019	Date and time - Representations for information interchange – Part 1: Basic rules	-	-
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/	-	-
SSL/TLS: RFC 5246	-	The TLS Protocol Version 1.2 http://tools.ietf.org/html/rfc5246.txt	-	-
X.509 v3: ISO/IEC-9594-8	-	Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks	-	-
HTTP: RFC 2616	-	Hypertext Transfer Protocol – HTTP/1.1 http://www.ietf.org/rfc/rfc2616.txt	-	-

<u>Publication</u>	<u>Year</u>	<u>Title</u>	<u>EN/HD</u>	<u>Year</u>
HTTPS: RFC 2818	-	HTTP Over TLS http://www.ietf.org/rfc/rfc2818.txt	-	-
Base64: RFC 3548	-	The Base16, Base32, and Base64 Data Encodings http://www.ietf.org/rfc/rfc3548.txt	-	-
X690: ISO/IEC 8825-1	-	Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)	-	-
IEEE-754	-	Standard for Floating-Point Arithmetic	-	-
HMAC	-	HMAC – Keyed-Hashing for Message Authentication http://www.ietf.org/rfc/rfc2104.txt	-	-
PKCS #1:	-	PKCS #1 – RSA Cryptography Specifications Version 2.0 http://www.ietf.org/rfc/rfc2437.txt	-	-
PKCS #12	-	PKCS #12 – Personal Information Exchange Syntax v1.1 http://www.ietf.org/rfc/rfc7292.txt	-	-
FIPS 180-4	-	Secure Hash Standard (SHS) https://csrc.nist.gov/publications/detail/fips/180/4/final	-	-
FIPS 197	-	Advanced Encryption Standard (AES) https://csrc.nist.gov/publications/detail/fips/197/final	-	-
UTF-8	-	UTF-8, a transformation format of ISO 10646 http://www.ietf.org/rfc/rfc3629.txt	-	-
RFC 3280	-	RFC 3280 – X.509 Public Key Infrastructure Certificate and CRL Profile http://www.ietf.org/rfc/rfc3280.txt	-	-
RFC 4514	-	RFC 4514 – LDAP: String Representation of Distinguished Names http://www.ietf.org/rfc/rfc4514.txt	-	-
NTP	-	RFC 1305 – Network Time Protocol (Version 3) Specification, Implementation and Analysis http://www.ietf.org/rfc/rfc1305.txt	-	-
Kerberos	-	Web Services Security – Kerberos Token Profile 1.1 http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-KerberosTokenProfile.pdf	-	-
RFC 1738	-	RFC 1738 – Uniform Resource Locators (URL) http://www.ietf.org/rfc/rfc1738.txt	-	-
RFC 2141	-	RFC 2141 – URN Syntax http://www.ietf.org/rfc/rfc2141.txt	-	-
RFC 6455	-	RFC 6455 – The WebSocket Protocol http://www.ietf.org/rfc/rfc6455.txt	-	-

<u>Publication</u>	<u>Year</u>	<u>Title</u>	<u>EN/HD</u>	<u>Year</u>
RFC 7523	-	JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants https://tools.ietf.org/rfc/rfc7523.txt	-	-
RFC 6749	-	The OAuth 2.0 Authorization Framework http://www.ietf.org/rfc/rfc6749.txt	-	-
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	-	RFC 6960 – X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP https://www.ietf.org/rfc/rfc6960.txt	-	-

This page deliberately left blank

CONTENTS

FOREWORD	8
1 Scope	11
2 Normative references	11
3 Terms, definitions, abbreviated terms and symbols.....	13
3.1 Terms and definitions.....	13
3.2 Abbreviated terms and symbols	14
4 Overview	14
5 Data encoding	16
5.1 General.....	16
5.1.1 Overview	16
5.1.2 Built-in Types	16
5.1.3 Guid	17
5.1.4 ByteString.....	17
5.1.5 ExtensionObject	17
5.1.6 Variant.....	18
5.1.7 Decimal	18
5.2 OPC UA Binary	19
5.2.1 General	19
5.2.2 Built-in Types	19
5.2.3 Decimal	30
5.2.4 Enumerations	30
5.2.5 Arrays.....	30
5.2.6 Structures.....	31
5.2.7 Structures with optional fields	33
5.2.8 Unions	35
5.2.9 Messages	36
5.3 OPC UA XML.....	37
5.3.1 Built-in Types	37
5.3.2 Decimal	43
5.3.3 Enumerations	43
5.3.4 Arrays.....	44
5.3.5 Structures.....	44
5.3.6 Structures with optional fields	45
5.3.7 Unions	45
5.3.8 Messages	46
5.4 OPC UA JSON.....	46
5.4.1 General	46
5.4.2 Built-in Types	46
5.4.3 Decimal	52
5.4.4 Enumerations	52
5.4.5 Arrays.....	52
5.4.6 Structures.....	53
5.4.7 Structures with optional fields	53
5.4.8 Unions	54
5.4.9 Messages	54
6 Message SecurityProtocols	55

6.1	Security handshake	55
6.2	Certificates	56
6.2.1	General	56
6.2.2	Application Instance Certificate.....	57
6.2.3	Certificate Chains	58
6.3	Time synchronization	58
6.4	UTC and International Atomic Time (TAI)	58
6.5	Issued User Identity Tokens.....	58
6.5.1	Kerberos.....	58
6.5.2	JSON Web Token (JWT).....	59
6.5.3	OAuth2	60
6.6	WS Secure Conversation	62
6.7	OPC UA Secure Conversation	62
6.7.1	Overview	62
6.7.2	MessageChunk structure	62
6.7.3	MessageChunks and error handling.....	67
6.7.4	Establishing a SecureChannel	67
6.7.5	Deriving keys.....	69
6.7.6	Verifying Message security	70
7	TransportProtocols	71
7.1	OPC UA Connection Protocol.....	71
7.1.1	Overview	71
7.1.2	Message structure	72
7.1.3	Establishing a connection	75
7.1.4	Closing a connection	77
7.1.5	Error handling.....	77
7.2	OPC UA TCP	79
7.3	SOAP/HTTP.....	79
7.4	OPC UA HTTPS.....	79
7.4.1	Overview	79
7.4.2	Session-less Services.....	81
7.4.3	XML Encoding	81
7.4.4	OPC UA Binary Encoding	82
7.4.5	JSON Encoding	82
7.5	WebSockets.....	83
7.5.1	Overview	83
7.5.2	Protocol Mapping.....	84
7.5.3	Security	84
7.6	Well known addresses	85
8	Normative Contracts	86
8.1	OPC Binary Schema	86
8.2	XML Schema and WSDL.....	86
8.3	Information Model Schema.....	86
8.4	Formal definition of UA Information Model.....	86
8.5	Constants	86
8.6	DataType encoding	86
8.7	Security configuration	86
Annex A (normative)	Constants.....	87
A.1	Attribute Ids	87

A.2	Status Codes	87
A.3	Numeric Node Ids	88
Annex B (normative)	OPC UA Nodeset	89
Annex C (normative)	Type declarations for the OPC UA native Mapping	90
Annex D (normative)	WSDL for the XML Mapping	91
D.1	XML Schema	91
D.2	WSDL Port Types	91
D.3	WSDL Bindings	91
Annex E (normative)	Security settings management	92
E.1	Overview	92
E.2	SecuredApplication	93
E.3	CertificateIdentifier	96
E.4	CertificateStoreIdentifier	98
E.5	CertificateList	99
E.6	CertificateValidationOptions	99
Annex F (normative)	Information Model XML Schema	101
F.1	Overview	101
F.2	UANodeSet	101
F.3	UANode	103
F.4	Reference	104
F.5	RolePermission	104
F.6	UAType	104
F.7	UAInstance	105
F.8	UAVariable	105
F.9	UAMethod	106
F.10	TranslationType	106
F.11	UADatatype	107
F.12	DatatypeDefinition	108
F.13	DatatypeField	108
F.14	Variant	109
F.15	Example	110
F.16	UANodeSetChanges	112
F.17	NodesToAdd	113
F.18	ReferencesToChange	113
F.19	ReferenceToChange	114
F.20	NodesToDelete	114
F.21	NodeToDelete	114
F.22	UANodeSetChangesStatus	115
F.23	NodeSetStatusList	115
F.24	NodeSetStatus	115
Bibliography		117
Figure 1 – The OPC UA Stack Overview		15
Figure 2 – Encoding Integers in a binary stream		20
Figure 3 – Encoding Floating Points in a binary stream		20
Figure 4 – Encoding Strings in a binary stream		21
Figure 5 – Encoding GUIDs in a binary stream		22

Figure 6 – Encoding XmlElement in a binary stream	22
Figure 7 – A String NodeId.....	23
Figure 8 – A Two Byte NodeId	24
Figure 9 – A Four Byte NodeId.....	24
Figure 10 – Security handshake.....	55
Figure 11 – OPC UA Secure Conversation MessageChunk.....	63
Figure 12 – OPC UA Connection Protocol Message structure	72
Figure 13 – Client initiated OPC UA Connection Protocol connection.....	76
Figure 14 – Server initiated OPC UA Connection Protocol connection.....	76
Figure 15 – Closing a OPC UA Connection Protocol connection	77
Figure 16 – Scenarios for the HTTPS Transport.....	80
Figure 17 – Setting up Communication over a WebSocket	84
Table 1 – Built-in Data Types.....	16
Table 2 – Guid structure	17
Table 3 – Layout of Decimal	19
Table 4 – Supported Floating Point Types.....	20
Table 5 – NodeId components	22
Table 6 – NodeId DataEncoding values	23
Table 7 – Standard NodeId Binary DataEncoding.....	23
Table 8 – Two Byte NodeId Binary DataEncoding	24
Table 9 – Four Byte NodeId Binary DataEncoding.....	24
Table 10 – ExpandedNodeId Binary DataEncoding	25
Table 11 – DiagnosticInfo Binary DataEncoding.....	26
Table 12 – QualifiedName Binary DataEncoding	26
Table 13 – LocalizedText Binary DataEncoding	27
Table 14 – Extension Object Binary DataEncoding.....	28
Table 15 – Variant Binary DataEncoding.....	29
Table 16 – Data Value Binary DataEncoding.....	30
Table 17 – Sample OPC UA Binary Encoded structure.....	32
Table 18 – Sample OPC UA Binary Encoded Structure with optional fields	34
Table 19 – Sample OPC UA Binary Encoded Structure	35
Table 20 – XML Data Type Mappings for Integers.....	37
Table 21 – XML Data Type Mappings for Floating Points	37
Table 22 – Components of NodeId	39
Table 23 – Components of ExpandedNodeId	40
Table 24 – Components of Enumeration	44
Table 25 – JSON Object Definition for a NodeId	48
Table 26 – JSON Object Definition for an ExpandedNodeId	49
Table 27 – JSON Object Definition for a StatusCode	49
Table 28 – JSON Object Definition for a DiagnosticInfo	50
Table 29 – JSON Object Definition for a QualifiedName.....	50
Table 30 – JSON Object Definition for a LocalizedText.....	50

Table 31 – JSON Object Definition for an ExtensionObject	51
Table 32 – JSON Object Definition for a Variant	51
Table 33 – JSON Object Definition for a DataValue	52
Table 34 – JSON Object Definition for a Decimal	52
Table 35 – JSON Object Definition for a <i>Structure</i> with Optional Fields	53
Table 36 – JSON Object Definition for a Union	54
Table 37 – SecurityPolicy	56
Table 38 – Application Instance Certificate	57
Table 39 – Kerberos UserTokenPolicy	59
Table 40 – JWT UserTokenPolicy	59
Table 41 – JWT IssuerEndpointUrl Definition	60
Table 42 – Access Token Claims	61
Table 43 – OPC UA Secure Conversation Message header	63
Table 44 – Asymmetric algorithm Security header	64
Table 45 – Symmetric algorithm Security header	65
Table 46 – Sequence header	65
Table 47 – OPC UA Secure Conversation Message footer	66
Table 48 – OPC UA Secure Conversation Message abort body	67
Table 49 – OPC UA Secure Conversation OpenSecureChannel Service	68
Table 50 – PRF inputs for RSA based SecurityPolicies	70
Table 51 – Cryptography key generation parameters	70
Table 52 – OPC UA Connection Protocol Message header	72
Table 53 – OPC UA Connection Protocol Hello Message	73
Table 54 – OPC UA Connection Protocol Acknowledge Message	74
Table 55 – OPC UA Connection Protocol Error Message	74
Table 56 – OPC UA Connection Protocol ReverseHello Message	75
Table 57 – OPC UA Connection Protocol error codes	78
Table 58 – WebSocket Protocols Mappings	84
Table 59 – Well known addresses for Local Discovery Servers	85
Table A.1 – Identifiers assigned to Attributes	87
Table E.1 – SecuredApplication	94
Table E.2 – CertificateIdentifier	97
Table E.3 – Structured directory store	98
Table E.4 – CertificateStoreIdentifier	99
Table E.5 – CertificateList	99
Table E.6 – CertificateValidationOptions	100
Table F.1 – UANodeSet	102
Table F.2 – UANode	103
Table F.3 – Reference	104
Table F.4 – RolePermission	104
Table F.5 – UANodeSet Type Nodes	104
Table F.6 – UANodeSet Instance Nodes	105
Table F.7 – UAInstance	105

Table F.8 – UVariable	106
Table F.9 – UAMethod	106
Table F.10 – TranslationType	107
Table F.11 – UADatatype	108
Table F.12 – DataTypeDefinition	108
Table F.13 – DataTypeField	109
Table F.14 – UANodeSetChanges	112
Table F.15 – NodesToAdd	113
Table F.16 – ReferencesToChange	113
Table F.17 – ReferencesToChange	114
Table F.18 – NodesToDelete	114
Table F.19 – ReferencesToChange	114
Table F.20 – UANodeSetChangesStatus	115
Table F.21 – NodeSetStatusList	115
Table F.22 – NodeSetStatus	116

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 third edition cancels and replaces the second edition published in 2015. This edition constitutes a technical revision.

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

a) Encodings:

- added JSON encoding for PubSub (non-reversible);
- added JSON encoding for Client/Server (reversible);
- added support for optional fields in structures;
- added support for Unions.

b) Transport mappings:

- added WebSocket secure connection – WSS;
- added support for reverse connectivity;
- added support for session-less service invocation in HTTPS.

c) Deprecated Transport (missing support on most platforms):

- SOAP/HTTP with WS-SecureConversation (all encodings).

d) Added mapping for JSON Web Token.

e) Added support for Unions to NodeSet Schema.

f) Added batch operations to add/delete nodes to/from NodeSet Schema.

g) Added support for multi-dimensional arrays outside of Variants.

h) Added binary representation for Decimal data types.

i) Added mapping for an OAuth2 Authorization Framework.

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

FDIS	Report on voting
65E/718/FDIS	65E/734/RVD

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 IEC 62541, certain document conventions are used:

Italics are used to denote a defined term or definition that appears in Clause 3 in one of the parts of the 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 also, 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 –

Part 6: Mappings

1 Scope

This part of IEC 62541 specifies the OPC Unified Architecture (OPC UA) mapping between the security model described in IEC TR 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 TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts*

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

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

ISO 8601-1:2019, *Date and time – Representations for information interchange – Part 1: Basic rules*

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/>

SSL/TLS: RFC 5246 – The TLS Protocol Version 1.2
<http://tools.ietf.org/html/rfc5246.txt>

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*

HTTP: RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1
<http://www.ietf.org/rfc/rfc2616.txt>

HTTPS: RFC 2818 – HTTP Over TLS
<http://www.ietf.org/rfc/rfc2818.txt>

Base64: RFC 3548 – The Base16, Base32, and Base64 Data Encodings
<http://www.ietf.org/rfc/rfc3548.txt>

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

IEEE-754: Standard for Floating-Point Arithmetic

HMAC: HMAC – Keyed-Hashing for Message Authentication
<http://www.ietf.org/rfc/rfc2104.txt>

PKCS #1: PKCS #1 – RSA Cryptography Specifications Version 2.0
<http://www.ietf.org/rfc/rfc2437.txt>

PKCS #12: PKCS #12 – Personal Information Exchange Syntax v1.1
<http://www.ietf.org/rfc/rfc7292.txt>

FIPS 180-4: Secure Hash Standard (SHS)
<https://csrc.nist.gov/publications/detail/fips/180/4/final>

FIPS 197: Advanced Encryption Standard (AES)
<https://csrc.nist.gov/publications/detail/fips/197/final>

UTF-8: UTF-8, a transformation format of ISO 10646
<http://www.ietf.org/rfc/rfc3629.txt>

RFC 3280: RFC 3280 – X.509 Public Key Infrastructure Certificate and CRL Profile
<http://www.ietf.org/rfc/rfc3280.txt>

RFC 4514: RFC 4514 – LDAP: String Representation of Distinguished Names
<http://www.ietf.org/rfc/rfc4514.txt>

NTP: RFC 1305 – Network Time Protocol (Version 3) Specification, Implementation and Analysis
<http://www.ietf.org/rfc/rfc1305.txt>

Kerberos: Web Services Security – Kerberos Token Profile 1.1
<http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-KerberosTokenProfile.pdf>

RFC 1738: RFC 1738 – Uniform Resource Locators (URL)
<http://www.ietf.org/rfc/rfc1738.txt>

RFC 2141: RFC 2141 – URN Syntax
<http://www.ietf.org/rfc/rfc2141.txt>

RFC 6455: RFC 6455 – The WebSocket Protocol
<http://www.ietf.org/rfc/rfc6455.txt>

RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format
<http://www.ietf.org/rfc/rfc7159.txt>

RFC 7523: JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants
<https://tools.ietf.org/rfc/rfc7523.txt>

RFC 6749: The OAuth 2.0 Authorization Framework
<http://www.ietf.org/rfc/rfc6749.txt>

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: RFC 6960 – X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP
<https://www.ietf.org/rfc/rfc6960.txt>

3 Terms, definitions, abbreviated terms and symbols

3.1 Terms and definitions

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

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at <http://www.electropedia.org/>
- ISO Online browsing platform: available at <http://www.iso.org/obp>

3.1.1

CertificateDigest

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

DevelopmentPlatform

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

3.1.4

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.

3.1.5

SecurityProtocol

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

3.1.6**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.

3.1.7**TransportConnection**

full-duplex communication link established between OPC UA applications

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

3.1.8**TransportProtocol**

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

3.2 Abbreviated terms and symbols

API	application programming interface
ASN.1	Abstract Syntax Notation #1 (used in X690)
CSV	comma separated value (file format)
ECC	elliptic curve cryptography
HTTP	Hypertext Transfer Protocol
HTTPS	Secure Hypertext Transfer Protocol
IPSec	Internet Protocol Security
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 SSL/TLS)
TCP	Transmission Control Protocol
TLS	Transport Layer Security (defined in SSL/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 IEC 62541 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 TR 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 5, 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, the Java programming language does not support an unsigned integer which means that any Java 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.

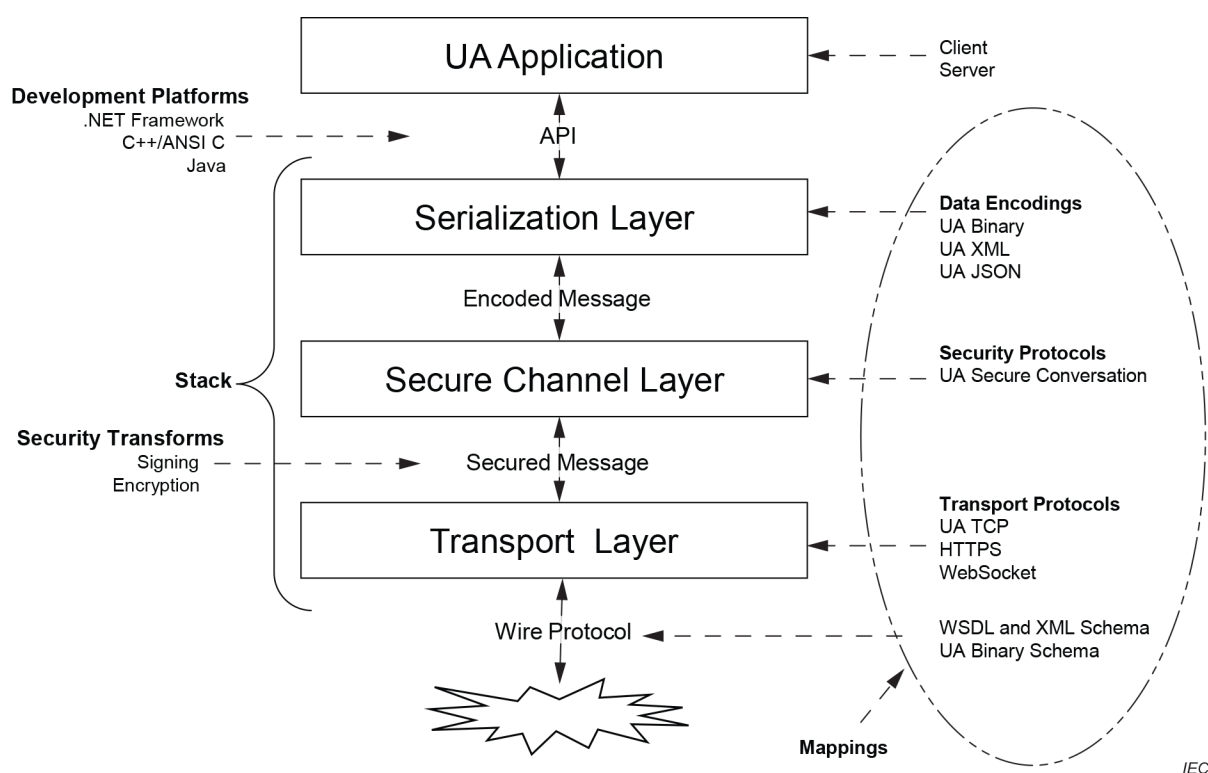


Figure 1 – The OPC UA Stack Overview

The layers described in this document 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 data encodings: 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	Description
1	Boolean	A two-state logical value (true or false).
2	SByte	An integer value between –128 and 127 inclusive.
3	Byte	An integer value between 0 and 255 inclusive.
4	Int16	An integer value between –32 768 and 32 767 inclusive.
5	UInt16	An integer value between 0 and 65 535 inclusive.
6	Int32	An integer value between –2 147 483 648 and 2 147 483 647 inclusive.
7	UInt32	An integer value between 0 and 4 294 967 295 inclusive.
8	Int64	An integer value between –9 223 372 036 854 775 808 and 9 223 372 036 854 775 807 inclusive.
9	UInt64	An integer value between 0 and 18 446 744 073 709 551 615 inclusive.
10	Float	An IEEE single precision (32 bit) floating point value.
11	Double	An IEEE double precision (64 bit) floating point value.
12	String	A sequence of Unicode characters.
13	DateTime	An instance in time.
14	Guid	A 16-byte value that can be used as a globally unique identifier.
15	ByteString	A sequence of octets.
16	XmlElement	An XML element.
17	NodeId	An identifier for a node in the address space of an OPC UA <i>Server</i> .
18	ExpandedNodeId	A NodeId that allows the namespace URI to be specified instead of an index.
19	StatusCode	A numeric identifier for an error or condition that is associated with a value or an operation.
20	QualifiedName	A name qualified by a namespace.
21	LocalizedText	Human readable text with an optional locale identifier.
22	ExtensionObject	A structure that contains an application specific data type that may not be recognized by the receiver.
23	DataValue	A data value with an associated status code and timestamps.
24	Variant	A union of all of the types specified above.
25	DiagnosticInfo	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.

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 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 need to represent *ByteStrings* as arrays of *SByte*. In this case, the requirements for *Byte* would then apply to *SByte*.

5.1.5 ExtensionObject

An *ExtensionObject* is a container for any *Structured DataTypes* which cannot be encoded as one of the other built-in data types. The *ExtensionObject* contains a complex value serialized as a sequence of bytes or as an XML element. It also contains an identifier which indicates what data it contains and how it is encoded.

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*.

Server implementers should use namespace qualified numeric *NodeIds* 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.

5.1.6 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*.

Variables with a *DataType* of *BaseDataType* are mapped to a *Variant*, however, the *ValueRank* and *ArrayDimensions Attributes* place restrictions on what is allowed in the *Variant*. For example, if the *ValueRank* is *Scalar* then the *Variant* may only contain scalar values.

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.

5.1.7 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 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
TypeId	NodeId	The identifier for the <i>Decimal</i> <i>DataType</i> .
Encoding	Byte	This value is always 1.
Length	Int32	The length of the <i>Decimal</i> . If the length is less than or equal to 0 then the <i>Decimal</i> value is 0.
Scale	Int16	A signed integer representing the power of ten used to scale the 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	Byte [*]	A 2-complement signed integer representing the unscaled value. The number of bits is inferred from the length of the <i>length</i> field. If the number of bits is 0 then the value is 0. 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*.

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*.

5.2 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 default value 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.

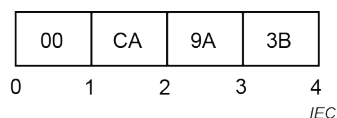


Figure 2 – Encoding Integers in a binary stream

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 4 lists the bit ranges for the supported floating point types.

Table 4 – Supported Floating Point Types

Name	Width (bits)	Fraction	Exponent	Sign
Float	32	0 to 22	23 to 30	31
Double	64	0 to 51	52 to 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 (000000000000F8FF) or (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.

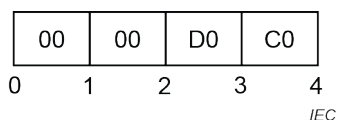


Figure 3 – Encoding Floating Points in a binary stream

5.2.2.4 String

All *String* values are encoded as a sequence of UTF-8 characters without a null terminator and 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.

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:

- a) 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.
- b) A date/time is encoded as the maximum value for an *Int64* if either
 - 1) The value is equal to or greater than 9999-12-31 11:59:59PM UTC,
 - 2) The value is the latest date that can be represented with the *DevelopmentPlatform's* encoding.
- c) A date/time is decoded as the earliest time that can be represented on the platform if either
 - 1) The encoded value is 0,
 - 2) The encoded value represents a time earlier than the earliest time that can be represented with the *DevelopmentPlatform's* encoding.
- d) A date/time is decoded as the latest time that can be represented on the platform if either
 - 1) The encoded value is the maximum value for an *Int64*,
 - 2) 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

IEC

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 fragment 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	72	6F	74	E6	B0	B4	3C	3F	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 the value of the *Variant* to the *StatusCode Bad_DecodingError*.

5.2.2.9 NodeId

The components of a *NodeId* are described the Table 5.

Table 5 – 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	*	The identifier for a node in the address space of an OPC UA Server.

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 6. Table 6 through Table 9 describe the structure of each possible format (they exclude the byte which indicates the format).

Table 6 – 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 7. The standard *DataEncoding* is used for all formats that do not have an explicit format defined.

Table 7 – 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.

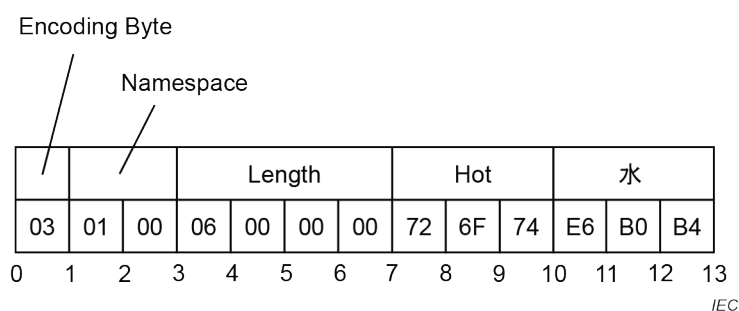


Figure 7 – A String NodeId

The Two Byte *NodeId DataEncoding* has the structure shown in Table 8.

Table 8 – 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.

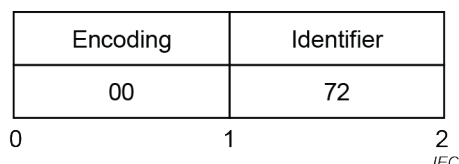


Figure 8 – A Two Byte NodeId

The Four Byte *NodeId DataEncoding* has the structure shown in Table 9.

Table 9 – 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.

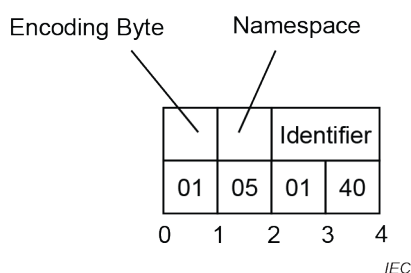


Figure 9 – A Four Byte NodeId

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 10.

Table 10 – ExpandedNodeId Binary DataEncoding

Name	Data Type	Description
NodeId	NodeId	The NamespaceUri and ServerIndex flags in the NodeId encoding indicate whether those fields are present in the stream.
NamespaceUri	String	Not present if null or Empty.
ServerIndex	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 11 – 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 12.

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

Table 12 – 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 13 – 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* and the number of bytes encoded.

An *ExtensionObject* may be encoded by the application which means it is passed as a *ByteString* or an *XmlElement* 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 14.

Table 14 – 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 a <i>XmlElement</i> .
Length	Int32	The length of the object body. The length shall be specified if the body is encoded.
Body	Byte [*]	The object body. 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.

ExtensionObjects are used in two contexts: as values contained in *Variant* structures or as parameters in OPC UA Messages.

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

5.2.2.16 Variant

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

The structure of a *Variant* is shown in Table 15.

Table 15 – 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 Array Dimensions 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> and pass both onto the application. Encoders shall not use these IDs.
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]
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>.

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

5.2.2.17 DataValue

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

The fields of a *DataValue* are described in Table 16.

Table 16 – 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 Source Timestamp is <i>DateTime.MinValue</i>. 0x08 False if the Server Timestamp is <i>DateTime.MinValue</i>. 0x10 False if the Source PicoSeconds is 0. 0x20 False if the Server PicoSeconds is 0.
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 <i>Server</i> 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 <i>Server</i> timestamp is missing the picoseconds are ignored.

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

5.2.3 Decimal

Decimals are encoded as described in 5.1.7.

A *Decimal* does not have a NULL value.

5.2.4 Enumerations

Enumerations are encoded as *Int32* values.

An *Enumeration* does not have a NULL value.

5.2.5 Arrays

One dimensional *Arrays* are encoded as a sequence of elements preceded by the number of elements encoded as an *Int32* value. If an *Array* is null, then its length is encoded as –1. An *Array* of zero length is different from an *Array* that is null so encoders and decoders shall preserve this distinction.

Multi-dimensional *Arrays* are encoded as an *Int32 Array* containing the dimensions followed by a list of all the values in the *Array*. The total number of values is equal to the product of the dimensions. The number of values is 0 if one or more dimensions are less than or equal to 0. The process for reconstructing the multi-dimensional array is described in 5.2.2.16.

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 default values 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;
};
```

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.

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 17 which would make the total length 37 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*.