

BS EN 62541-6:2015



BSI Standards Publication

# OPC unified architecture

Part 6: Mappings

**bsi.**

...making excellence a habit.™

### **National foreword**

This British Standard is the UK implementation of EN 62541-6:2015. It is identical to IEC 62541-6:2015. It supersedes BS EN 62541-6:2011 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee AMT/7, Industrial communications: process measurement and control, including fieldbus.

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

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 2015.

Published by BSI Standards Limited 2015

ISBN 978 0 580 83005 1

ICS 25.040.40; 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 June 2015.

### **Amendments/corrigenda issued since publication**

<b>Date</b>	<b>Text affected</b>
-------------	----------------------

---

EUROPEAN STANDARD

**EN 62541-6**

NORME EUROPÉENNE

EUROPÄISCHE NORM

May 2015

ICS 25.040.40; 35.100

Supersedes EN 62541-6:2011

English Version

**OPC unified architecture - Part 6: Mappings  
(IEC 62541-6:2015)**Architecture unifiée OPC - Partie 6: Correspondances  
(IEC 62541-6:2015)OPC Unified Architecture - Teil 6: Protokollabbildungen  
(IEC 62541-6:2015)

This European Standard was approved by CENELEC on 2015-04-29. 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, Former Yugoslav Republic of Macedonia, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, the Netherlands, Norway, Poland, Portugal, Romania, 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: Avenue Marnix 17, B-1000 Brussels**

## Foreword

The text of document 65E/377/CDV, future edition 2 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 62541-6:2015.

The following dates are fixed:

- latest date by which the document has to be implemented at national level by publication of an identical national standard or by endorsement (dop) 2016-01-29
- latest date by which the national standards conflicting with the document have to be withdrawn (dow) 2018-04-29

This document supersedes EN 62541-6:2011.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. CENELEC [and/or CEN] 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, and supports essential requirements of EU Directive(s).

## Endorsement notice

The text of the International Standard IEC 62541-6:2015 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, 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.

NOTE 1 When 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](http://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	EN 62541-3	-
IEC 62541-4	-	OPC Unified Architecture - Part 4: Services	EN 62541-4	-
IEC 62541-5	-	OPC unified architecture - Part 5: Information Model	EN 62541-5	-
IEC 62541-7	-	OPC unified architecture - Part 7: Profiles	EN 62541-7	-
IEEE 754	2008	IEEE Standard for Binary Floating-Point Arithmetic	-	-
ITU-T X.509	-	Information technology - Open systems interconnection - The Directory: Public-key and attribute certificate frameworks	-	-
ITU-T X.690	2002	Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)	-	-
FIPS PUB 180-2	2002	Secure Hash Standard	-	-
FIPS PUB 197	2001	Advanced Encryption Standard (AES)	-	-
RFC 1305	1992	Network Time Protocol (Version 3) - Specification, Implementation and Analysis	-	-
RFC 2104	1997	HMAC: Keyed-Hashing for Message Authentication	-	-
RFC 2437	1998	PKCS #1: RSA Cryptography Specifications Version 2.0	-	-

<u>Publication</u>	<u>Year</u>	<u>Title</u>	<u>EN/HD</u>	<u>Year</u>
RFC 2616	1999	Hypertext Transfer Protocol - HTTP/1.1	-	-
RFC 3280	2002	Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile	-	-
RFC 3548	2003	The Base16, Base32, and Base64 Data Encodings	-	-
RFC 3629	2003	UTF-8, a transformation format of ISO 10646	-	-
RFC 4514	2006	Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names	-	-
RFC 5246	2008	The Transport Layer Security (TLS) Protocol Version 1.2	-	-
SOAP Part 1	2007	SOAP Version 1.2 - Part 1: Messaging Framework	-	-
SOAP Part 2	2007	SOAP Version 1.2 - Part 2: Adjuncts	-	-
WS-Addressing	2004	Web Services Addressing (WS-Addressing)	-	-
XML Encryption	2002	XML Encryption Syntax and Processing	-	-
XML Schema Part 1	2004	XML Schema - Part 1: Structures	-	-
XML Schema Part 2	2004	XML Schema - Part 2: Datatypes	-	-
XML Signature	2008	XML Signature Syntax and Processing	-	-

## CONTENTS

FOREWORD.....	7
1 Scope.....	9
2 Normative references .....	9
3 Terms, definitions, abbreviations and symbols.....	11
3.1 Terms and definitions.....	11
3.2 Abbreviations and symbols .....	11
4 Overview .....	12
5 Data encoding .....	13
5.1 General.....	13
5.1.1 Overview .....	13
5.1.2 Built-in Types.....	13
5.1.3 Guid.....	14
5.1.4 ByteString.....	15
5.1.5 ExtensionObject` .....	15
5.1.6 Variant.....	15
5.2 OPC UA Binary .....	16
5.2.1 General.....	16
5.2.2 Built-in Types.....	16
5.2.3 Enumerations.....	25
5.2.4 Arrays .....	25
5.2.5 Structures .....	25
5.2.6 Messages .....	26
5.3 XML.....	26
5.3.1 Built-in Types.....	26
5.3.2 Enumerations.....	33
5.3.3 Arrays.....	33
5.3.4 Structures .....	33
5.3.5 Messages .....	34
6 Message SecurityProtocols .....	34
6.1 Security handshake .....	34
6.2 Certificates .....	35
6.2.1 General.....	35
6.2.2 Application Instance Certificate.....	36
6.2.3 Signed Software Certificate.....	36
6.3 Time synchronization .....	37
6.4 UTC and International Atomic Time (TAI).....	37
6.5 Issued User Identity Tokens – Kerberos.....	38
6.6 WS Secure Conversation .....	38
6.6.1 Overview .....	38
6.6.2 Notation.....	40
6.6.3 Request Security Token (RST/SCT).....	40
6.6.4 Request Security Token Response (RSTR/SCT).....	41
6.6.5 Using the SCT .....	42
6.6.6 Cancelling Security contexts.....	42
6.7 OPC UA Secure Conversation .....	43
6.7.1 Overview .....	43

6.7.2	MessageChunk structure .....	43
6.7.3	MessageChunks and error handling .....	46
6.7.4	Establishing a SecureChannel .....	47
6.7.5	Deriving keys .....	48
6.7.6	Verifying Message Security .....	49
7	Transport Protocols .....	50
7.1	OPC UA TCP .....	50
7.1.1	Overview .....	50
7.1.2	Message structure .....	50
7.1.3	Establishing a connection .....	52
7.1.4	Closing a connection .....	53
7.1.5	Error handling .....	54
7.1.6	Error recovery .....	54
7.2	SOAP/HTTP .....	56
7.2.1	Overview .....	56
7.2.2	XML Encoding .....	56
7.2.3	OPC UA Binary Encoding .....	57
7.3	HTTPS .....	57
7.3.1	Overview .....	57
7.3.2	XML Encoding .....	59
7.3.3	OPC UA Binary Encoding .....	60
7.4	Well known addresses .....	60
8	Normative Contracts .....	61
8.1	OPC Binary Schema .....	61
8.2	XML Schema and WSDL .....	61
Annex A (normative)	Constants .....	62
A.1	Attribute Ids .....	62
A.2	Status Codes .....	62
A.3	Numeric Node Ids .....	62
Annex B (normative)	OPC UA Nodeset .....	64
Annex C (normative)	Type declarations for the OPC UA native Mapping .....	65
Annex D (normative)	WSDL for the XML Mapping .....	66
D.1	XML Schema .....	66
D.2	WDSL Port Types .....	66
D.3	WSDL Bindings .....	66
Annex E (normative)	Security settings management .....	67
E.1	Overview .....	67
E.2	SecuredApplication .....	68
E.3	CertificateIdentifier .....	71
E.4	CertificateStoreIdentifier .....	73
E.5	CertificateList .....	73
E.6	CertificateValidationOptions .....	73
Annex F (normative)	Information Model XML Schema .....	75
F.1	Overview .....	75
F.2	UANodeSet .....	75
F.3	UANode .....	76
F.4	Reference .....	76
F.5	UAType .....	77



F.6	UAInstance .....	77
F.7	UAVariable .....	77
F.8	UAMethod.....	78
F.9	TranslationType .....	78
F.10	UADataType .....	79
F.11	DataTypeDefintion .....	79
F.12	DataTypeInfoField .....	80
F.13	Variant.....	80
F.14	Example (Informative).....	81
Figure 1	– The OPC UA Stack Overview .....	13
Figure 2	– Encoding Integers in a binary stream .....	16
Figure 3	– Encoding Floating Points in a binary stream.....	17
Figure 4	– Encoding Strings in a binary stream .....	17
Figure 5	– Encoding GuidS in a binary stream.....	18
Figure 6	– Encoding XmlElements in a binary stream.....	19
Figure 7	– A String NodeId.....	20
Figure 8	– A Two Byte NodeId .....	20
Figure 9	– A Four Byte NodeId.....	21
Figure 10	– Security handshake.....	34
Figure 11	– Relevant XML Web Services specifications .....	39
Figure 12	– The WS Secure Conversation handshake.....	39
Figure 13	– OPC UA Secure Conversation MessageChunk.....	43
Figure 14	– OPC UA TCP Message structure.....	52
Figure 15	– Establishing a OPC UA TCP connection.....	53
Figure 16	– Closing a OPC UA TCP connection .....	53
Figure 17	– Recovering an OPC UA TCP connection .....	55
Figure 18	– Scenarios for the HTTPS Transport.....	58
Table 1	– Built-in Data Types.....	14
Table 2	– Guid structure .....	14
Table 3	– Supported Floating Point Types.....	17
Table 4	– NodeId components .....	19
Table 5	– NodeId DataEncoding values .....	19
Table 6	– Standard NodeId Binary DataEncoding.....	19
Table 7	– Two Byte NodeId Binary DataEncoding .....	20
Table 8	– Four Byte NodeId Binary DataEncoding.....	20
Table 9	– ExpandedNodeId Binary DataEncoding .....	21
Table 10	– DiagnosticInfo Binary DataEncoding.....	22
Table 11	– QualifiedName Binary DataEncoding .....	22
Table 12	– LocalizedText Binary DataEncoding .....	22
Table 13	– Extension Object Binary DataEncoding.....	23
Table 14	– Variant Binary DataEncoding.....	24
Table 15	– Data Value Binary DataEncoding.....	25

Table 16 – Sample OPC UA Binary Encoded structure.....	26
Table 17 – XML Data Type Mappings for Integers.....	27
Table 18 – XML Data Type Mappings for Floating Points .....	27
Table 19 – Components of NodeId .....	29
Table 20 – Components of ExpandedNodeId .....	30
Table 21 – Components of Enumeration .....	33
Table 22 – SecurityPolicy .....	35
Table 23 – ApplicationInstanceCertificate .....	36
Table 24 – SignedSoftwareCertificate .....	37
Table 25 – Kerberos UserTokenPolicy .....	38
Table 26 – WS-* Namespace prefixes .....	40
Table 27 – RST/SCT Mapping to an OpenSecureChannel Request.....	41
Table 28 – RSTR/SCT Mapping to an OpenSecureChannel Response.....	42
Table 29 – OPC UA Secure Conversation Message header .....	44
Table 30 – Asymmetric algorithm Security header.....	44
Table 31 – Symmetric algorithm Security header .....	45
Table 32 – Sequence header .....	45
Table 33 – OPC UA Secure Conversation Message footer .....	46
Table 34 – OPC UA Secure Conversation Message abort body.....	47
Table 35 – OPC UA Secure Conversation OpenSecureChannel Service .....	47
Table 36 – Cryptography key generation parameters .....	49
Table 37 – OPC UA TCP Message header.....	50
Table 38 – OPC UA TCP Hello Message.....	51
Table 39 – OPC UA TCP Acknowledge Message .....	51
Table 40 – OPC UA TCP Error Message.....	52
Table 41 – OPC UA TCP error codes .....	54
Table 42 – WS-Addressing headers .....	56
Table 43 – Well known addresses for Local Discovery Servers .....	60
Table A.1 – Identifiers assigned to Attributes .....	62
Table E.1 – SecuredApplication .....	69
Table E.2 – CertificateIdentifier.....	71
Table E.3 – Structured directory store.....	72
Table E.4 – CertificateStoreIdentifier .....	73
Table E.5 – CertificateList.....	73
Table E.6 – CertificateValidationOptions .....	74
Table F.1 – UANodeSet .....	75
Table F.2 – UANode .....	76
Table F.3 – Reference .....	77
Table F.4 – UANodeSet Type Nodes.....	77
Table F.5 – UANodeSet Instance Nodes .....	77
Table F.6 – UAInstance .....	77
Table F.7 – UAVariable.....	78
Table F.8 – UAMethod .....	78

Table F.9 – TranslationType .....	79
Table F.10 – UADDataType.....	79
Table F.11 – DataTypeDefinition.....	80
Table F.12 – DataTypeField.....	80

## 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 second edition cancels and replaces the first edition published in 2011. This edition constitutes a technical revision.

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

- a) Some applications need to operation in environments with no access to cryptography libraries. To support this a new HTTPS transport has been defined in 7.3;
- b) The padding byte is not long enough to handle asymmetric key sizes larger than 2048 bits. Added an additional padding byte to 6.7.2 to handle this case.
- c) Fixed errors in SOAP action URIs defined in 7.2.2;

- d) Needed a standard way to serialize nodes in an address space. Added the UANodeSet schema defined in Annex F;

The text of this standard is based on the following documents:

CDV	Report on voting
65E/377/CDV	65E/405/RVC

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

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

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 publication will remain unchanged until the stability date indicated on the IEC web site under "<http://webstore.iec.ch>" in the data related to the specific publication. At this date, the publication 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, described 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*

XML Schema Part 1: XML Schema Part 1: Structures

<http://www.w3.org/TR/xmlschema-1/>

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/>

SOAP Part 2: SOAP Version 1.2 Part 2: Adjuncts

<http://www.w3.org/TR/soap12-part2/>

XML Encryption: XML Encryption Syntax and Processing

<http://www.w3.org/TR/xmlenc-core/>

XML Signature: XML-Signature Syntax and Processing

<http://www.w3.org/TR/xmldsig-core/>

WS Security: SOAP Message Security 1.1

<http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>

WS Addressing: Web Services Addressing (WS-Addressing)

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

WS Trust: WS Trust 1.3

<http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.html>

WS Secure Conversation: WS Secure Conversation 1.3

<http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html>

WS Security Policy: WS Security Policy 1.2

<http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html>

SSL/TLS: RFC 5246 – The TLS Protocol Version 1.2

<http://tools.ietf.org/html/rfc5246.txt>

X509: X.509 Public Key Certificate Infrastructure

<http://www.itu.int/rec/T-REC-X.509-200003-I/e>

WS-I Basic Profile 1.1: WS-I Basic Profile Version 1.1

<http://www.ws-i.org/Profiles/BasicProfile-1.1.html>

WS-I Basic Security Profile 1.1: WS-I Basic Security Profile Version 1.1

<http://www.ws-i.org/Profiles/BasicSecurityProfile-1.1.html>

HTTP: RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1

<http://www.ietf.org/rfc/rfc2616.txt>

Base64: RFC 3548 – The Base16, Base32, and Base64 Data Encodings

<http://www.ietf.org/rfc/rfc3548.txt>

X690: ITU-T X.690 – Basic (BER), Canonical (CER) and Distinguished (DER) Encoding Rules

<http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf>

IEEE-754: Standard for Binary Floating-Point Arithmetic

<http://grouper.ieee.org/groups/754/>

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>

FIPS 180-2: Secure Hash Standard (SHA)

<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>

FIPS 197: Advanced Encryption Standard (AES)

<http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

UTF8: UTF-8, a transformation format of ISO 10646

<http://tools.ietf.org/html/rfc3629>

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)

<http://www.ietf.org/rfc/rfc1305.txt>

Kerberos: WS Security Kerberos Token Profile 1.1

<http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-KerberosTokenProfile.pdf>

### 3 Terms, definitions, abbreviations 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 and IEC 62541-3 as well as the following apply.

##### 3.1.1

##### **DataEncoding**

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

##### 3.1.2

##### **Mapping**

specifies 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.

##### 3.1.3

##### **Security Protocol**

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

##### 3.1.4

##### **Stack Profile**

a 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.

##### 3.1.5

##### **Transport Protocol**

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

#### 3.2 Abbreviations and symbols

API	Application Programming Interface
ASN.1	Abstract Syntax Notation #1 (used in X690)
BP	WS-I Basic Profile Version
BSP	WS-I Basic Security Profile
CSV	Comma Separated Value (File Format)
HTTP	Hypertext Transfer Protocol
HTTPS	Secure Hypertext Transfer Protocol
IPSec	Internet Protocol Security
RST	Request Security Token
OID	Object Identifier (used with ASN.1)
RSTR	Request Security Token Response



SCT	Security Context Token
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)
UTF8	Unicode Transformation Format (8-bit) (Defined in UTF8)
UA	Unified Architecture
UASC	OPC UA Secure Conversation
WS-*	XML Web Services Specifications
WSS	WS Security
WS-SC	WS Secure Conversation
XML	Extensible Markup Language

#### 4 Overview

Other parts of this series of standards 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.

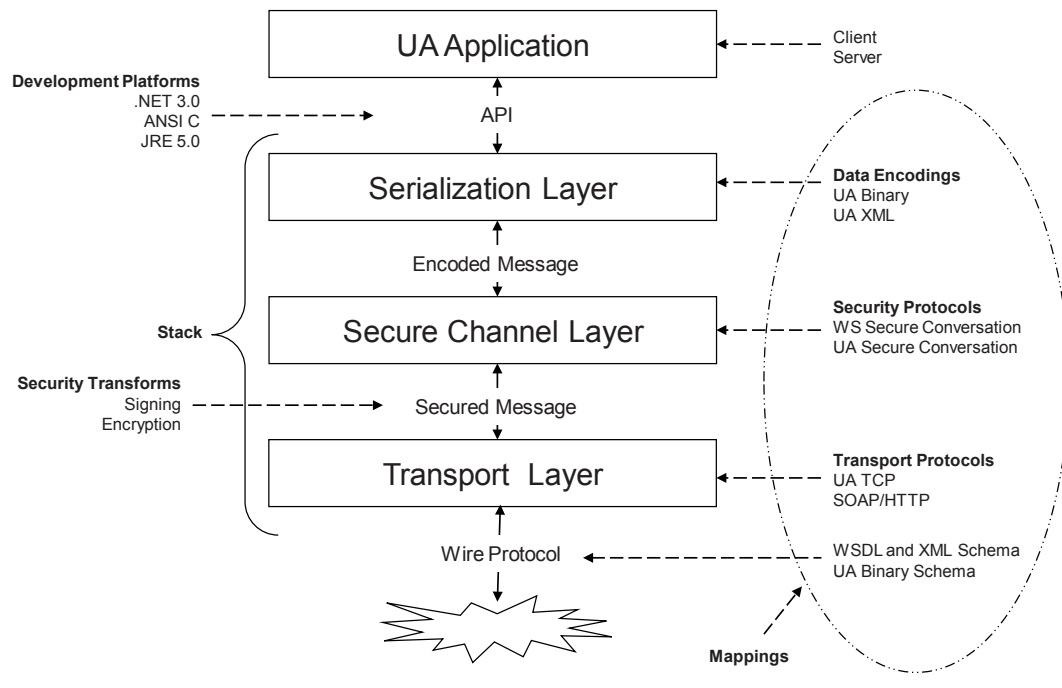
This standard defines *Mappings* between the abstract specifications and technologies that can be used to implement them. The *Mappings* are organized into three groups: *DataEncodings*, *SecurityProtocols* and *TransportProtocols*. Different *Mappings* are combined together to create *StackProfiles*. 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 standard defines the *DataEncodings* in Clause 5, the *SecurityProtocols* in Clause 6 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 appendices.

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 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 standard.



IEC

**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 standard defines two data encodings: OPC UA Binary and OPC UA XML. 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.
3	Byte	An integer value between 0 and 256.
4	Int16	An integer value between –32 768 and 32 767.
5	UInt16	An integer value between 0 and 65 535.
6	Int32	An integer value between –2 147 483 648 and 2 147 483 647.
7	UInt32	An integer value between 0 and 429 4967 295.
8	Int64	An integer value between –9 223 372 036 854 775 808 and 9 223 372 036 854 775 807
9	UInt64	An integer value between 0 and 18 446 744 073 709 551 615.
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 Server.
18	ExpandedNodeId	A NodeId that allows the namespace URI to be specified instead of an index.
19	StatusCode	A numeric identifier for a 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 standard. In addition, this standard 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 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 have 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 *Complex Data* types 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.

*Complex Data* types are represented in a *Server* address space as sub-types of the *Structure DataType*. The *DataEncodings* available for any given *Complex Data* type 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 *Complex Data* values into *ExtensionObjects*.

#### 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 *Development Platforms* 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.

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

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

*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.

## 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 *Complex Data* 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

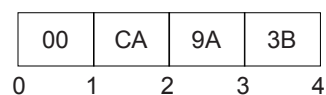
A *Boolean* value shall be encoded as a single byte where a value of 0 (zero) is false and any non-zero value is true.

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) should be encoded as a 32 bit integer in the stream.



IEC

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

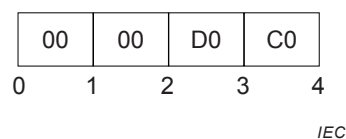
**Table 3 – Supported Floating Point Types**

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) should be 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.

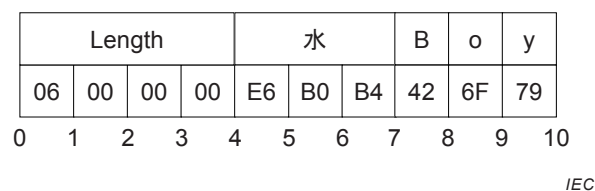
**Figure 3 – Encoding Floating Points in a binary stream**

#### 5.2.2.4 String

All *String* values are encoded as a sequence of UTF8 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” should be encoded in a byte stream.

**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 Clause 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.
  - 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 9999-01-01 11:59:59PM,
  - 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" should be 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 UTF8 string and then encoded as *ByteString*.

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



Length				<A>			Hot			水			</A>				
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

IEC

Figure 6 – Encoding XmlElements in a binary stream

### 5.2.2.9 Nodeld

The components of a *Nodeld* are described the Table 4.

Table 4 – Nodeld components

Name	Data Type	Description
Namespace	UInt16	The index for a namespace URI. An index of 0 is used for OPC UA defined <i>Nodelds</i> .
IdentifierType	Enum	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 <i>Server</i> .

The *DataEncoding* of a *Nodeld* 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 *Nodeld*. The possible *DataEncoding* formats are shown in Table 5. The tables that follow describe the structure of each possible format (they exclude the byte which indicates the format).

Table 5 – Nodeld 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 <i>String</i> value.
Guid	0x04	A <i>Guid</i> value.
ByteString	0x05	An opaque ( <i>ByteString</i> ) value.
NamespaceUri Flag	0x80	See discussion of <i>ExpandedNodeld</i> in 5.2.2.10.
ServerIndex Flag	0x40	See discussion of <i>ExpandedNodeld</i> in 5.2.2.10.

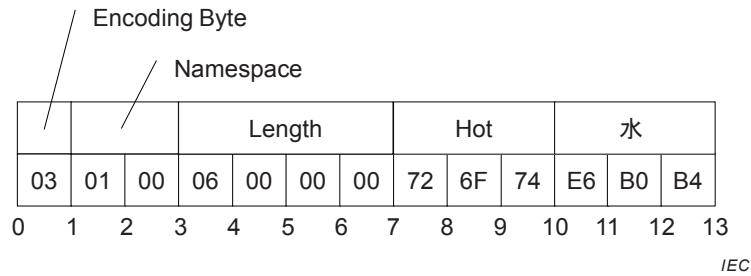
The standard *Nodeld DataEncoding* has the structure shown in Table 6. The standard *DataEncoding* is used for all formats that do not have an explicit format defined.

Table 6 – Standard Nodeld Binary DataEncoding

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



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



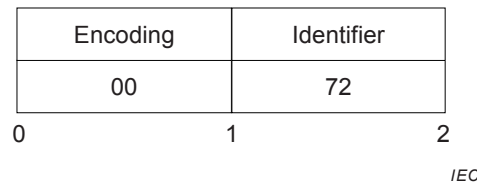
**Figure 7 – A String NodeId**

The Two Byte *NodeId DataEncoding* has the structure shown in Table 7.

**Table 7 – 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.



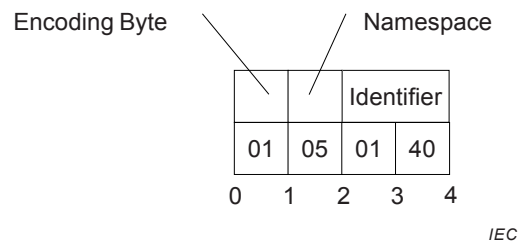
**Figure 8 – A Two Byte NodeId**

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

**Table 8 – 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 9.

**Table 9 – ExpandedNodeid Binary DataEncoding**

Name	Data Type	Description
Nodeid	Nodeid	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.

**Table 10 – 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.
LocalizedText	Int32	A human readable summary of the status code.
Locale	Int32	The locale used for the localized text.
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 11.

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

**Table 11 – 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 12 – 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 *DataTypeId* 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 *EncodingMask* 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 13.

**Table 13 – 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.1.
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.

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

#### 5.2.2.16 Variant

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

The structure of a *VARIANT* is shown in Table 14.

**Table 14 – Variant Binary DataEncoding**

Name	Data Type	Description
EncodingMask	Byte	The type of data encoded in the stream. 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.
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	Int32[]	The length of each dimension. This field is only present if the array dimensions flag is set in the encoding mask. The lower rank dimensions appear first in the array.

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 15.

**Table 15 – 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 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.

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 Enumerations

Enumerations are encoded as *Int32* values.

### 5.2.4 Arrays

*Arrays* that occur outside of a *VARIANT* 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 can only be encoded within a *VARIANT*.

### 5.2.5 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 complex type shall be encoded.

*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++ syntax:

```
class Type2
{
    int A;
    int B;
};

class Type1
{
    int X;
    int NoOfY;
    Type2* Y;
    int Z;
};
```

The Y field is a pointer to an array with a length stored in NoOfY.

An instance of *Type1* which contains an array of two *Type2* instances would be encoded as 37 byte sequence. If the instance of *Type1* was encoded in an *ExtensionObject* it would have the encoded form shown in Table 16. 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*.

**Table 16 – Sample OPC UA Binary Encoded structure**

Field	Bytes	Value
Type Id	4	The identifier for Type1
Encoding	1	0x1 for ByteString
Length	4	28
X	4	The value of field 'X'
NoOfY	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'

## 5.2.6 Messages

*Messages* are encoded as *ExtensionObjects*. The parameters in each *Message* are serialized in the same way the fields of a *Structure* are serialized. The *TypeId* field contains the *DataTypeEncoding* identifier for the *Message*. The *Length* field is omitted since the *Messages* are defined by this series of OPC UA standards.

Each OPC UA *Service* described in IEC 62541-4 has a request and response *Message*. The *DataTypeEncoding* IDs assigned to each *Service* are given in A.3.

## 5.3 XML

### 5.3.1 Built-in Types

#### 5.3.1.1 General

Most built-in types are encoded in XML using the formats defined in XML Schema Part 2 specification. Any special restrictions or usages are discussed below. Some of the built-in types have an XML Schema defined for them using the syntax defined in XML Schema Part 1.

The prefix *xs:* is used to denote a symbol defined by the XML Schema specification.

### 5.3.1.2 Boolean

A Boolean value is encoded as an *xs:boolean* value.

### 5.3.1.3 Integer

Integer values are encoded using one of the subtypes of the *xs:decimal* type. The mappings between the OPC UA integer types and XML schema data types are shown in Table 17.

**Table 17 – XML Data Type Mappings for Integers**

Name	XML Type
SByte	xs:byte
Byte	xs:unsignedByte
Int16	xs:short
UInt16	xs:unsignedShort
Int32	xs:int
UInt32	xs:unsignedInt
Int64	xs:long
UInt64	xs:unsignedLong

### 5.3.1.4 Floating Point

Floating point values are encoded using one of the XML floating point types. The mappings between the OPC UA floating point types and XML schema data types are shown in Table 18.

**Table 18 – XML Data Type Mappings for Floating Points**

Name	XML Type
Float	xs:float
Double	xs:double

The XML floating point type supports positive infinity (INF), negative infinity (-INF) and not-a-number (NaN).

### 5.3.1.5 String

A *String* value is encoded as an *xs:string* value.

### 5.3.1.6 DateTime

A *DateTime* value is encoded as an *xs:dateTime* value.

All *DateTime* values shall be encoded as UTC times or with the time zone explicitly specified.

Correct:

```
2002-10-10T00:00:00+05:00
2002-10-09T19:00:00Z
```

Incorrect:

```
2002-10-09T19:00:00
```

It is recommended that all *xs:dateTime* values be represented in UTC format.

The earliest and latest date/time values that can be represented on a *DevelopmentPlatform* have special meaning and shall not be literally encoded in XML.



The earliest date/time value on a *DevelopmentPlatform* shall be encoded in XML as '0001-01-01T00:00:00Z'.

The latest date/time value on a *DevelopmentPlatform* shall be encoded in XML as '9999-12-31T11:59:59Z'

If a decoder encounters a *xs:dateTime* value that cannot be represented on the *DevelopmentPlatform* it should convert the value to either the earliest or latest date/time that can be represented on the *DevelopmentPlatform*. The XML decoder should not generate an error if it encounters an out of range date value.

The earliest date/time value on a *DevelopmentPlatform* is equivalent to a null date/time value.

### 5.3.1.7 Guid

A *Guid* is encoded using the string representation defined in 5.1.3.

The XML schema for a *Guid* is:

```
<xs:complexType name="Guid">
  <xs:sequence>
    <xs:element name="String" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

### 5.3.1.8 ByteString

A *ByteString* value is encoded as an *xs:base64Binary* value (see Base64).

The XML schema for a *ByteString* is:

```
<xs:element name="ByteString" type="xs:base64Binary" nillable="true"/>
```

### 5.3.1.9 XmlElement

An *XmlElement* value is encoded as an *xs:complexType* with the following XML schema:

```
<xs:complexType name="XmlElement">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="1" processContents="lax" />
  </xs:sequence>
</xs:complexType>
```

*XmlElement*s may only be used inside *Variant* or *ExtensionObject* values.

### 5.3.1.10 NodeId

A *NodeId* value is encoded as an *xs:string* with the syntax:

```
ns=<namespaceindex>;<type>=<value>
```

The elements of the syntax are described in Table 19.

**Table 19 – Components of NodeId**

Field	Data Type	Description
<namespaceindex>	UInt16	The <i>NamespaceIndex</i> formatted as a base 10 number. If the index is 0 then the entire 'ns=0;' clause shall be omitted.
<type>	Enum	A flag that specifies the <i>IdentifierType</i> . The flag has the following values: i      NUMERIC (UInteger) s      STRING (String) g      GUID (Guid) b      OPAQUE (ByteString)
<value>	*	The <i>Identifier</i> encoded as string. The <i>Identifier</i> is formatted using the XML data type mapping for the <i>IdentifierType</i> . Note that the <i>Identifier</i> may contain any non-null UTF8 character including whitespace.

Examples of *NodeIds*:

```
i=13
ns=10;i=-1
ns=10;s>Hello:World
g=09087e75-8e5e-499b-954f-f2a9603db28a
ns=1;b=M/RbKBSRVkePCePcx24oRA==
```

The XML schema for a *NodeId* is:

```
<xs:complexType name="NodeId">
  <xs:sequence>
    <xs:element name="Identifier" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

### 5.3.1.11 ExpandedNodeId

An *ExpandedNodeId* value is encoded as an xs:string with the syntax:

```
svr=<serverindex>;ns=<namespaceindex>;<type>=<value>
or
svr=<serverindex>;nsu=<uri>;<type>=<value>
```

The possible fields are shown in Table 20.

**Table 20 – Components of ExpandedNodeId**

Field	Data Type	Description
<serverindex>	UInt32	The <i>ServerIndex</i> formatted as a base 10 number. If the <i>ServerIndex</i> is 0 then the entire 'svr=0;' clause shall be omitted.
<namespaceindex>	UInt16	The <i>NamespaceIndex</i> formatted as a base 10 number. If the <i>NamespaceIndex</i> is 0 then the entire 'ns=0;' clause shall be omitted. The <i>NamespaceIndex</i> shall not be present if the URI is present.
<uri>	String	The <i>NamespaceUri</i> formatted as a string. Any reserved characters in the URI shall be replaced with a '%' followed by its 8 bit ANSI value encoded as two hexadecimal digits (case insensitive). For example, the character ';' would be replaced by '%3B'. The reserved characters are ';' and '%'. If the <i>NamespaceUri</i> is null or empty then 'nsu=;' clause shall be omitted.
<type>	Enum	A flag that specifies the <i>IdentifierType</i> . This field is described in Table 19.
<value>	*	The <i>Identifier</i> encoded as string. This field is described in Table 19.

The XML schema for an *ExpandedNodeId* is:

```
<xs:complexType name="ExpandedNodeId">
  <xs:sequence>
    <xs:element name="Identifier" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

#### 5.3.1.12 StatusCode

A *StatusCode* is encoded as an *xs:unsignedInt* with the following XML schema:

```
<xs:complexType name="StatusCode">
  <xs:sequence>
    <xs:element name="Code" type="xs:unsignedInt" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

#### 5.3.1.13 DiagnosticInfo

An *DiagnosticInfo* value is encoded as an *xs:complexType* with the following XML schema:

```
<xs:complexType name="DiagnosticInfo">
  <xs:sequence>
    <xs:element name="SymbolicId" type="xs:int" minOccurs="0" />
    <xs:element name="NamespaceUri" type="xs:int" minOccurs="0" />
    <xs:element name="LocalizedText" type="xs:int" minOccurs="0"/>
    <xs:element name="Locale" type="xs:int" minOccurs="0"/>
    <xs:element name="AdditionalInfo" type="xs:string" minOccurs="0"/>
    <xs:element name="InnerStatusCode" type="tns:StatusCode"
      minOccurs="0" />
    <xs:element name="InnerDiagnosticInfo" type="tns:DiagnosticInfo"
      minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

#### 5.3.1.14 QualifiedName

A *QualifiedName* value is encoded as an *xs:complexType* with the following XML schema:

```
<xs:complexType name="QualifiedName">
  <xs:sequence>
```

```

    <xs:element name="NamespaceIndex" type="xs:int" minOccurs="0" />
    <xs:element name="Name" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

```

### 5.3.1.15 LocalizedText

A *LocalizedText* value is encoded as an *xs:complexType* with the following XML schema:

```

<xs:complexType name="LocalizedText">
  <xs:sequence>
    <xs:element name="Locale" type="xs:string" minOccurs="0" />
    <xs:element name="Text" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

```

### 5.3.1.16 ExtensionObject

An *ExtensionObject* value is encoded as an *xs:complexType* with the following XML schema:

```

<xs:complexType name="ExtensionObject">
  <xs:sequence>
    <xs:element name="TypeId" type="tns:NodeId" minOccurs="0" />
    <xs:element name="Body" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:any minOccurs="0" processContents="lax"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

```

The body of the *ExtensionObject* contains a single element which is either a *ByteString* or XML encoded *Structure*. A decoder can distinguish between the two by inspecting the top level element. An element with the name *tns:ByteString* contains an OPC UA Binary encoded body. Any other name shall contain an OPC UA XML encoded body.

The *TypeId* is the *NodeId* for the *DataTypeEncoding Object*.

### 5.3.1.17 Variant

A *Variant* value is encoded as an *xs:complexType* with the following XML schema:

```

<xs:complexType name="Variant">
  <xs:sequence>
    <xs:element name="Value" minOccurs="0" nillable="true">
      <xs:complexType>
        <xs:sequence>
          <xs:any minOccurs="0" processContents="lax"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

```

If the *Variant* represents a scalar value then it shall contain a single child element with the name of the built-in type. For example, the single precision floating point value 3,141 5 would be encoded as:

```
<tns:Float>3.1415</tns:Float>
```

If the *Variant* represents a single dimensional array then it shall contain a single child element with the prefix 'ListOf' and the name built-in type. For example an *Array* of strings would be encoded as:

```
<tns:ListOfString>
  <tns:String>Hello</tns:String>
  <tns:String>World</tns:String>
</tns:ListOfString>
```

If the *Variant* represents a multidimensional *Array* then it shall contain a child element with the name '*Matrix*' with the two sub-elements shown in this example:

```
<tns:Matrix>
  <tns:Dimensions>
    <tns:Int32>2</tns:Int32>
    <tns:Int32>2</tns:Int32>
  </tns:Dimensions>
  <tns:Elements>
    <tns:String>A</tns:String>
    <tns:String>B</tns:String>
    <tns:String>C</tns:String>
    <tns:String>D</tns:String>
  </tns:Elements>
</tns:Matrix>
```

In this example, the array has the following elements:

```
[0,0] = "A"; [0,1] = "B"; [1,0] = "C"; [1,1] = "D"
```

The elements of a multi-dimensional *Array* are always flattened into a single dimensional *Array* where the higher rank dimensions are serialized first. This single dimensional *Array* is encoded as a child of the 'Elements' element. The 'Dimensions' element is an *Array* of *Int32* values that specify the dimensions of the array starting with the lowest rank dimension. The multi-dimensional *Array* can be reconstructed by using the dimensions encoded.

The complete set of built-in type names is found in Table 1.

### 5.3.1.18 DataValue

A *DataValue* value is encoded as a *xs:complexType* with the following XML schema:

```
<xs:complexType name="DataValue">
  <xs:sequence>
    <xs:element name="Value" type="tns:Variant" minOccurs="0"
      nillable="true" />
    <xs:element name="StatusCode" type="tns:StatusCode"
      minOccurs="0" />
    <xs:element name="SourceTimestamp" type="xs:dateTime"
      minOccurs="0" />
    <xs:element name="SourcePicoseconds" type="xs:unsignedShort"
      minOccurs="0"/>
    <xs:element name="ServerTimestamp" type="xs:dateTime"
      minOccurs="0" />
    <xs:element name="ServerPicoseconds" type="xs:unsignedShort"
      minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

### 5.3.2 Enumerations

*Enumerations* that are used as parameters in the *Messages* defined in IEC 62541-4 are encoded as *xs:string* with the following syntax:

```
<symbol>_<value>
```

The elements of the syntax are described in Table 21.

**Table 21 – Components of Enumeration**

Field	Type	Description
<symbol>	String	The symbolic name for the enumerated value.
<value>	UInt32	The numeric value associated with enumerated value.

For example, the XML schema for the *NodeClass* enumeration is:

```
<xs:simpleType name="NodeClass">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Unspecified_0" />
    <xs:enumeration value="Object_1" />
    <xs:enumeration value="Variable_2" />
    <xs:enumeration value="Method_4" />
    <xs:enumeration value="ObjectType_8" />
    <xs:enumeration value="VariableType_16" />
    <xs:enumeration value="ReferenceType_32" />
    <xs:enumeration value="DataType_64" />
    <xs:enumeration value="View_128" />
  </xs:restriction>
</xs:simpleType>
```

*Enumerations* that are stored in a *Variant* are encoded as an *Int32* value.

For example, any *Variable* could have a value with a *DataType* of *NodeClass*. In this case the corresponding numeric value is placed in the *Variant* (e.g. *NodeClass::Object* would be stored as a 1).

### 5.3.3 Arrays

*Array* parameters are always encoded by wrapping the elements in a container element and inserting the container into the structure. The name of the container element should be the name of the parameter. The name of the element in the array shall be the type name.

For example, the *Read* service takes an array of *ReadValueIds*. The XML schema would look like:

```
<xs:complexType name="ListOfReadValueId">
  <xs:sequence>
    <xs:element name="ReadValueId" type="tns:ReadValueId"
      minOccurs="0" maxOccurs="unbounded" nillable="true" />
  </xs:sequence>
</xs:complexType>
```

The nillable attribute shall be specified because XML encoders will drop elements in arrays if those elements are empty.

### 5.3.4 Structures

Structures are encoded as a *xs:complexType* with all of the fields appearing in a sequence. All fields are encoded as an *xs:element* and have *xs:maxOccurs* set to 1.

For example, the Read service has a *ReadValueId* structure in the request. The XML schema would look like:

```
<xs:complexType name="ReadValueId">
  <xs:sequence>
    <xs:element name="NodeId" type="tns:NodeId" minOccurs="1" />
    <xs:element name="AttributeId" type="xs:int" minOccurs="1" />
    <xs:element name="IndexRange" type="xs:string"
      minOccurs="0" nillable="true" />
    <xs:element name="DataEncoding" type="tns:NodeId" minOccurs="1" />
  </xs:sequence>
</xs:complexType>
```

### 5.3.5 Messages

*Messages* are encoded as an *xs:complexType*. The parameters in each *Message* are serialized in the same way the fields of a *Structure* are serialized.

## 6 Message SecurityProtocols

### 6.1 Security handshake

All *SecurityProtocols* shall implement the *OpenSecureChannel* and *CloseSecureChannel* services defined in IEC 62541-4. These *Services* specify how to establish a *SecureChannel* and how to apply security to *Messages* exchanged over that *SecureChannel*. The *Messages* exchanged and the security algorithms applied to them are shown in Figure 10.

*SecurityProtocols* shall support three *SecurityModes*: *None*, *Sign* and *SignAndEncrypt*. If the *SecurityMode* is *None* then no security is used and the security handshake shown in Figure 10 is not required. However, a *SecurityProtocol* implementation shall still maintain a logical channel and provide a unique identifier for the *SecureChannel*.

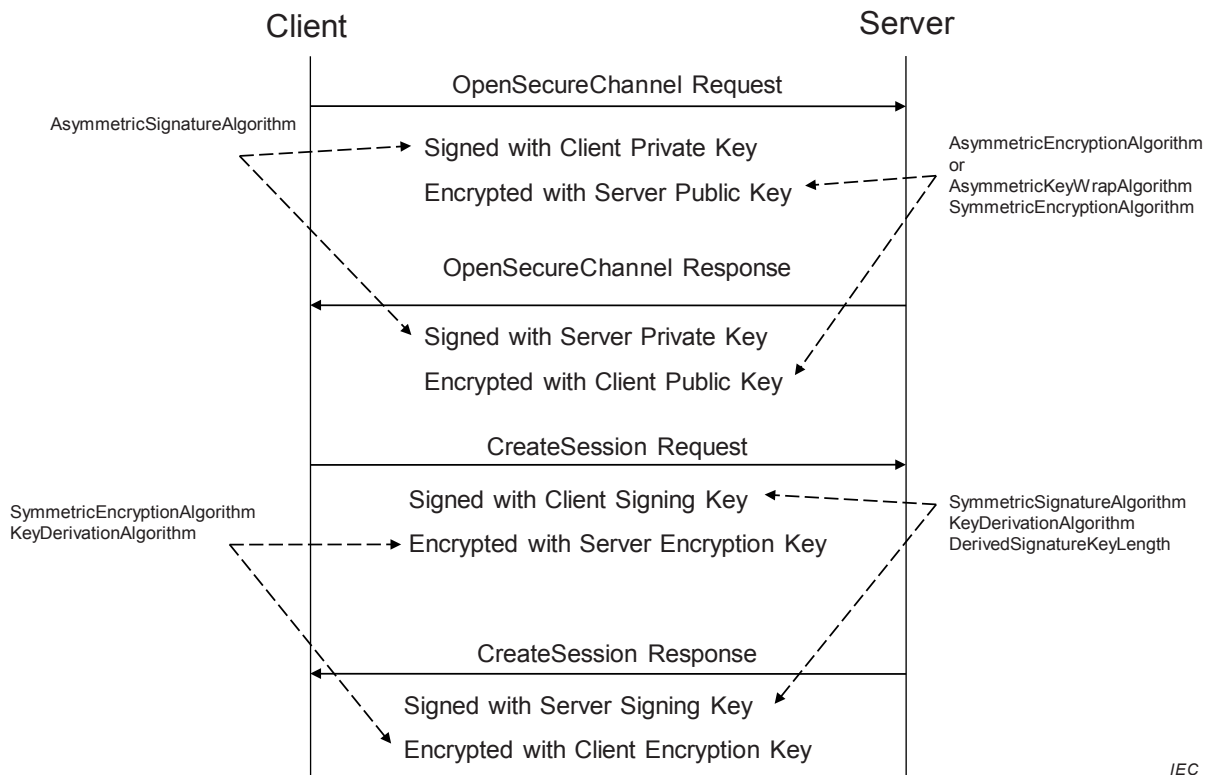


Figure 10 – Security handshake

Each *SecurityProtocol* mapping specifies exactly how to apply the security algorithms to the *Message*. A set of security algorithms that shall be used together during a security handshake is called a *SecurityPolicy*. IEC 62541-7 defines standard *SecurityPolicies* as parts of the standard *Profiles* which OPC UA applications are expected to support. IEC 62541-7 also defines a URI for each standard *SecurityPolicy*.

A *Stack* is expected to have built in knowledge of the *SecurityPolicies* that it supports. *Applications* specify the *SecurityPolicy* they wish to use by passing the URI to the *Stack*.

Table 22 defines the contents of a *SecurityPolicy*. Each *SecurityProtocol* mapping specifies how to use each of the parameters in the *SecurityPolicy*. A *SecurityProtocol* mapping may not make use of all of the parameters.

**Table 22 – SecurityPolicy**

Name	Description
PolicyUri	The URI assigned to the <i>SecurityPolicy</i> .
SymmetricSignatureAlgorithm	The URI of the symmetric signature algorithm to use.
SymmetricEncryptionAlgorithm	The URI of the symmetric key encryption algorithm to use.
AsymmetricSignatureAlgorithm	The URI of the asymmetric signature algorithm to use.
AsymmetricKeyWrapAlgorithm	The URI of the asymmetric key wrap algorithm to use.
AsymmetricEncryptionAlgorithm	The URI of the asymmetric key encryption algorithm to use.
MinAsymmetricKeyLength	The minimum length for an asymmetric key.
MaxAsymmetricKeyLength	The maximum length for an asymmetric key.
KeyDerivationAlgorithm	The key derivation algorithm to use.
DerivedSignatureKeyLength	The length in bits of the derived key used for <i>Message</i> authentication.

The *AsymmetricEncryptionAlgorithm* is used when encrypting the entire *Message* with an asymmetric key. Some *SecurityProtocols* do not encrypt the entire *Message* with an asymmetric key. Instead, they use the *AsymmetricKeyWrapAlgorithm* to encrypt a symmetric key and then use the *SymmetricEncryptionAlgorithm* to encrypt the *Message*.

The *AsymmetricSignatureAlgorithm* is used to sign a *Message* with an asymmetric key.

The *KeyDerivationAlgorithm* is used to create the keys used to secure *Messages* sent over the *SecureChannel*. The length of the keys used for encryption is implied by the *SymmetricEncryptionAlgorithm*. The length of the keys used for creating *Symmetric Signatures* depends on the *SymmetricSignatureAlgorithm* and may be different from the encryption key length.

## 6.2 Certificates

### 6.2.1 General

OPC UA *Applications* use *Certificates* to store the *Public Keys* needed for *Asymmetric Cryptography* operations. All *SecurityProtocols* use X509 Version 3 *Certificates* (see X509) encoded using the DER format (see X690). *Certificates* used by OPC UA *Applications* shall also conform to RFC 3280 which defines a profile for X509 *Certificates* when they are used as part of an Internet based *Application*.

The *ServerCertificate* and *ClientCertificate* parameters used in the abstract *OpenSecureChannel* service are instances of the *ApplicationInstance Certificate Data Type*. Subclause 6.2.2 describes how to create an X509 *Certificate* that can be used as an *ApplicationInstance Certificate*.

The *ServerSoftwareCertificates* and *ClientSoftwareCertificates* parameters in the abstract *CreateSession* and *ActivateSession Services* are instances of the *SignedSoftwareCertificate Data Type*. Subclause 6.2.3 describes how to create an X509 *Certificate* that can be used as a *SignedSoftwareCertificate*.



### 6.2.2 Application Instance Certificate

An *ApplicationInstanceCertificate* is a *ByteString* containing the DER encoded form (see X690) of an X509v3 *Certificate*. This *Certificate* is issued by certifying authority and identifies an instance of an *Application* running on a single host. The X509v3 fields contained in an *ApplicationInstance Certificate* are described in Table 23. The fields are defined completely in RFC 3280.

Table 23 also provides a mapping from the RFC 3280 terms to the terms used in the abstract definition of an *ApplicationInstanceCertificate* defined in IEC 62541-4.

**Table 23 – ApplicationInstanceCertificate**

Name	Part 4 Parameter Name	Description
ApplicationInstanceCertificate		An X509v3 <i>Certificate</i> .
version	version	shall be "V3"
serialNumber	serialNumber	The serial number assigned by the issuer.
signatureAlgorithm	signatureAlgorithm	The algorithm used to sign the <i>Certificate</i> .
signature	signature	The signature created by the Issuer.
issuer	issuer	The distinguished name of the <i>Certificate</i> used to create the signature. The <i>issuer</i> field is completely described in RFC 3280.
validity	validTo, validFrom	When the <i>Certificate</i> becomes valid and when it expires.
subject	subject	The distinguished name of the <i>Application Instance</i> . The Common Name attribute shall be specified and should be the <i>productName</i> or a suitable equivalent. The Organization Name attribute shall be the name of the Organization that executes the <i>Application</i> instance. This organization is usually not the vendor of the <i>Application</i> . Other attributes may be specified. The <i>subject</i> field is completely described in RFC 3280.
subjectAltName	applicationUri, hostnames	The alternate names for the <i>Application Instance</i> . Shall include a uniformResourceIdentifier which is equal to the <i>applicationUri</i> . <i>Servers</i> shall specify a <i>dnsName</i> or <i>IPAddress</i> which identifies the machine where the <i>Application Instance</i> runs. Additional <i>dnsNames</i> may be specified if the machine has multiple names. The <i>IPAddress</i> should not be specified if the <i>Server</i> has <i>dnsName</i> . The <i>subjectAltName</i> field is completely described in RFC 3280.
publicKey	publicKey	The public key associated with the <i>Certificate</i> .
keyUsage	keyUsage	Specifies how the <i>Certificate</i> key may be used. Shall include <i>digitalSignature</i> , <i>nonRepudiation</i> , <i>keyEncipherment</i> and <i>dataEncipherment</i> . Other key uses are allowed.
extendedKeyUsage	keyUsage	Specifies additional key uses for the <i>Certificate</i> . Shall specify <i>'serverAuth'</i> and/or <i>'clientAuth'</i> . Other key uses are allowed.
authorityKeyIdentifier		Provides more information about the key used to sign the <i>Certificate</i> . It shall be specified for <i>Certificates</i> signed by a CA. It should be specified for self-signed <i>Certificates</i> .

### 6.2.3 Signed Software Certificate

A *SignedSoftwareCertificate* is a *ByteString* containing the DER encoded form of an X509v3 *Certificate*. This *Certificate* is issued by a certifying authority and contains an X509v3 extension with the *SoftwareCertificate* which specifies the claims verified by the certifying authority. The X509v3 fields contained in a *SignedSoftwareCertificate* are described in Table 24. The fields are defined completely in RFC 3280.

**Table 24 – SignedSoftwareCertificate**

Name		Description
SignedSoftwareCertificate		An X509v3 <i>Certificate</i> .
version	version	Shall be "V3"
serialNumber	serialNumber	The serial number assigned by the issuer.
signatureAlgorithm	signatureAlgorithm	The algorithm used to sign the <i>Certificate</i> .
signature	signature	The signature created by the Issuer.
issuer	issuer	The distinguished name of the <i>Certificate</i> used to create the signature. The <i>issuer</i> field is completely described in RFC 3280.
validity	validTo, validFrom	When the <i>Certificate</i> becomes valid and when it expires.
subject	subject	The distinguished name of the product. The Common Name attribute shall be the same as the <i>productName</i> in the <i>SoftwareCertificate</i> and the Organization Name attribute shall be the <i>vendorName</i> in the <i>SoftwareCertificate</i> . Other attributes may be specified. The <i>subject</i> field is completely described in RFC 3280.
subjectAltName	productUri	The alternate names for the product. It shall include a 'uniformResourceIdentifier' which is equal to the <i>productUri</i> specified in the <i>SoftwareCertificate</i> . The <i>subjectAltName</i> field is completely described in RFC 3280.
publicKey	publicKey	The public key associated with the <i>Certificate</i> .
keyUsage	keyUsage	Specifies how the <i>Certificate</i> key may be used. shall be 'digitalSignature' and 'nonRepudiation' Other key uses are not allowed.
extendedKeyUsage	keyUsage	Specifies additional key uses for the <i>Certificate</i> . May specify 'codeSigning'. Other key usages are not allowed.
softwareCertificate	softwareCertificate	The XML encoded form of the <i>SoftwareCertificate</i> stored as UTF8 text. Subclause 5.3.4 describes how to encode a <i>SoftwareCertificate in XML</i> . The ASN.1 Object Identifier (OID) for this extension is: 1.2.840.113556.1.8000.2264.1.6.1

### 6.3 Time synchronization

All *Security Protocols* require that system clocks on communicating machines be reasonably synchronized in order to check the expiry times for *Certificates* or *Messages*. The amount of clock skew that can be tolerated depends on the system security requirements and *Applications* shall allow administrators to configure the acceptable clock skew when verifying times. A suitable default value is 5 minutes.

The Network Time Protocol (NTP) provides a standard way to synchronize a machine clock with a time server on the network. Systems running on a machine with a full featured operating system like Windows or Linux will already support NTP or an equivalent. Devices running embedded operating systems should support NTP.

If a device operating system cannot practically support NTP then an OPC UA *Application* can use the *Timestamps* in the *ResponseHeader* (see IEC 62541-4) to synchronize its clock. In this scenario the OPC UA *Application* will have to know the URL for a *Discovery Server* on a machine known to have the correct time. The OPC UA *Application* or a separate background utility would call the *FindServers Service* and set its clock to the time specified in the *ResponseHeader*. This process will need to be repeated periodically because clocks can drift over time.

### 6.4 UTC and International Atomic Time (TAI)

All times in OPC UA are in UTC, however, UTC can include discontinuities due to leap seconds or repeating seconds added to deal with variations in the earth's orbit and rotation. *Servers* that have access to source for International Atomic Time (TAI) may choose to use this instead of UTC. That said, *Clients* must always be prepared to deal with discontinuities due to the UTC or simply because the system clock is adjusted on the *Server* machine.

## 6.5 Issued User Identity Tokens – Kerberos

Kerberos *UserIdentityTokens* can be passed to the *Server* using the *IssuedIdentityToken*. The body of the token is an XML element that contains the WS-Security token as defined in the Kerberos Token Profile (Kerberos) specification.

*Servers* that support Kerberos authentication shall provide a *UserTokenPolicy* which specifies what version of the Kerberos Token Profile is being used, the Kerberos Realm and the Kerberos Principal Name for the *Server*. The Realm and Principal name are combined together with a simple syntax and placed in the *issuerEndpointUri* as shown in Table 25.

**Table 25 – Kerberos UserTokenPolicy**

Name	Description
tokenType	ISSUEDTOKEN_3
issuedTokenType	<a href="http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1">http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1</a>
issuerEndpointUri	A string with the form \\<realm>\<server principal name> where <realm> is the Kerberos realm name (e.g. Windows Domain); <server principal name> is the Kerberos principal name for the OPC UA Server.

The interface between the *Client* and *Server* applications and the Kerberos Authentication Service is application specific. The realm is the DomainName when using a Windows Domain controller as the Kerberos provider.

## 6.6 WS Secure Conversation

### 6.6.1 Overview

Any *Message* sent via SOAP may be secured with the WS Secure Conversation . This protocol specifies a way to negotiate shared secrets via WS Trust and then use these secrets to secure *Messages* exchanged with the mechanisms defined in WS Security.

The mechanisms for actually signing XML elements are described in the XML Signature specification. The mechanisms for encrypting XML elements are described in the XML Encryption specification.

WS Security Policy defines standard algorithm suites which can be used to secure SOAP *Messages*. These algorithm suites map directly onto the *SecurityPolicies* that are defined in IEC 62541-7. WS-I Basic Security Profile 1.1 defines best practices when using WS-Security which will help ensure interoperability. All OPC UA implementations shall conform to this specification.

The *Timestamp* header defined by WS Security is used to prevent replay attacks and shall be present and signed in all *Messages* exchanged.

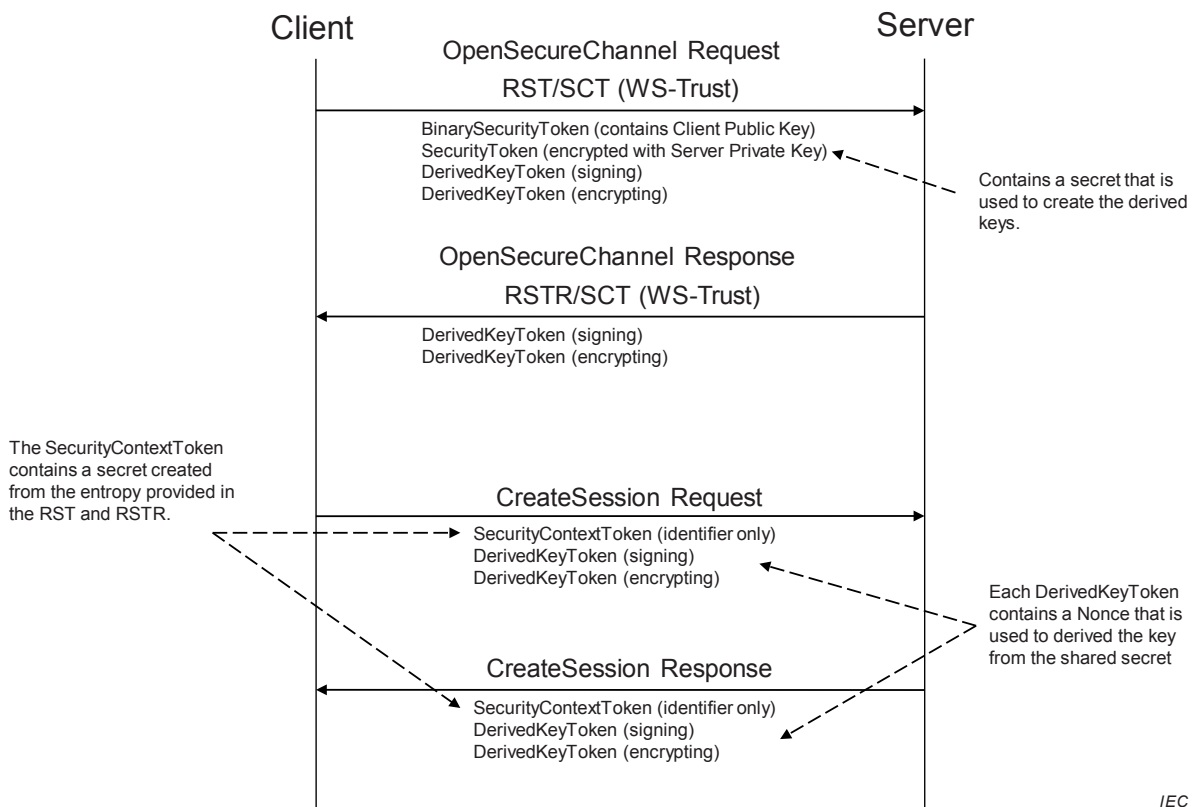
Figure 11 illustrates the relationship between the different WS-\* specifications that are used by this mapping. The versions of the WS-\* specifications shown in the diagram were the most current versions at the time of publication. IEC 62541-7 may define *Profiles* that require support for future versions of these specifications.

WS Secure Conversation 1.3			WS Security Policy 1.2
WS Security 1.1		WS Trust 1.3	
XML Signature 1.0	XML Encryption 1.0	WS Addressing 1.0	
SOAP 1.2			
HTTP or HTTPS (SSL/TLS)			

IEC

**Figure 11 – Relevant XML Web Services specifications**

Figure 12 illustrates how these WS-\* specifications are used in the security handshake.



IEC

**Figure 12 – The WS Secure Conversation handshake**

The RST (Request Security Token) and RSTR (Request Security Token Response) Messages are defined by WS Trust. WS Secure Conversation defines new actions for these Messages that tell the Server that the Client wants to create a SCT (Security Context Token). The SCT contains the shared keys that the Applications use to secure Messages sent over the SecureChannel.

Individual Messages are secured with keys derived from the SCT using the mechanism defined in WS Secure Conversation. The subclauses below specify the structure of the individual Messages and illustrate which features from the WS-\* specifications are required to implement the OPC UA security handshake.

### 6.6.2 Notation

SOAP *Messages* use XML elements defined in a number of different specifications. This document uses the prefixes in Table 26 to identify the specification that defines an XML element.

**Table 26 – WS-\* Namespace prefixes**

Prefix	Specification
wsu	WS-Security Utilities
wsse	WS-Security Extensions
wst	WS-Trust
wsc	WS-Secure Conversation
wsa	WS-Addressing
xenc	XML Encryption

### 6.6.3 Request Security Token (RST/SCT)

The Request Security Token *Messages* implements the abstract *OpenSecureChannel* request *Message* defined in IEC 62541-4. The syntax of this *Message* is defined by WS Trust. The structure of the *Message* is described in detail in WS Secure Conversation.

This *Message* shall have the following tokens:

- a) A wsse:BinarySecurityToken containing the *Client's Public Key*. The *Public Key* is sent in a DER encoded X509v3 *Certificate*.
- b) An encrypted wsse:SecurityToken containing *ClientNonce* used to derive keys. This *SecurityToken* shall be encrypted with the *AsymmetricKeyWrapAlgorithm* and the *Public Key* associated with the *Server's Application Instance Certificate*.
- c) A wsc:DerivedKeyToken which is used to sign the body, the WS Addressing headers and the wsu:Timestamp header using the *SymmetricSignatureAlgorithm*. The signature element shall then be signed using the *AsymmetricSignatureAlgorithm* with the *Client's Private Key*. The wsc:DerivedKeyToken shall also specify a *Nonce*.
- d) A wsc:DerivedKeyToken which is used to encrypt the body of the *Message* using the *SymmetricEncryptionAlgorithm*.

This *Message* shall have the wsa:Action, wsa:MessageId, wsa:ReplyTo and wsa:To headers defined by WS Addressing. The *Message* shall also have a wsu:Timestamp header defined by WS Security. These headers shall also be signed with the derived key used to sign the *Message* body.

The signature shall be calculated before applying encryption and the signature shall be encrypted.

The mapping between the *OpenSecureChannel* request parameters and the elements of the RST/SCT *Message* are shown in Table 27.

Table 27 – RST/SCT Mapping to an OpenSecureChannel Request

OpenSecureChannel Parameter	RST/SCT Element	Description
clientCertificate	wsse:BinarySecurityToken	Passed in the SOAP header.
requestType	wst:RequestType	Shall be "http://schemas.xmlsoap.org/ws/2005/02/trust/Issue" when creating a new SCT. Shall be "http://schemas.xmlsoap.org/ws/2005/02/trust/Renew" when renewing a SCT.
secureChannelId	wsse:SecurityTokenReference	Passed in the SOAP header when renewing an SCT.
securityMode securityPolicyUri	wst:SignatureAlgorithm wst:EncryptionAlgorithm wst:KeySize	These elements describe the <i>SecurityPolicy</i> requested by the <i>Client</i> . These elements shall match the <i>SecurityPolicy</i> used by the <i>Endpoint</i> that the <i>Client</i> wishes to connect to. These elements are optional.
clientNonce	wst:Entropy	This contains the <i>Nonce</i> specified by the <i>Client</i> . The <i>Nonce</i> is specified with the wst:BinarySecret element.
requestedLifetime	wst:Lifetime	The requested lifetime for the SCT. This element is optional.

#### 6.6.4 Request Security Token Response (RSTR/SCT)

The Request Security Token Response *Message* implements the abstract *OpenSecureChannel* response *Message* defined in IEC 62541-4. The syntax of this *Message* is defined by WS Trust. The use of the *Message* is described in detail in WS Secure Conversation. This *Message* not signed or encrypted with the asymmetric algorithms as described in IEC 62541-4. The symmetric algorithms and a key provided in the request *Message* are used instead.

This *Message* shall have the following tokens:

- a) A wsc:DerivedKeyToken which is used to sign the body, the WS Addressing headers and the wsu:Timestamp header using the *SymmetricSignatureAlgorithm*. This key is derived from the encrypted *SecurityToken* specified in the RST/SCT *Message*. The wsc:DerivedKeyToken shall also specify a *Nonce*.
- b) A wsc:DerivedKeyToken which is used to encrypt the body of the *Message* using the *SymmetricEncryptionAlgorithm*. This key is derived from the encrypted *SecurityToken* specified in the RST/SCT *Message*. The wsc:DerivedKeyToken shall also specify a *Nonce*.

This *Message* shall have the wsa:Action and wsa:RelatesTo headers defined by WS Addressing. The *Message* shall also have a wsu:Timestamp header defined by WS Security. These headers shall also be signed with the derived key used to sign the *Message* body.

The signature shall be calculated before applying encryption and the signature shall be encrypted.

The mapping between the *OpenSecureChannel* response parameters and the elements of the RSTR/SCT *Message* are shown Table 28.



**Table 28 – RSTR/SCT Mapping to an OpenSecureChannel Response**

OpenSecureChannel Parameter	RSTR/SCT Element	Description
---	wst:RequestedProofToken	This contains a wst:ComputedKey element which specifies the algorithm used to compute the shared secret key from the <i>Nonces</i> provided by the <i>Client</i> and the <i>Server</i> .
---	wst:TokenType	Specifies the type of <i>SecurityToken</i> issued.
securityToken	wst:RequestedSecurityToken	Specifies the new SCT (Security Context Token) or renewed SCT.
channelId	wsc:Identifier	An absolute URI which identifies the SCT.
tokenId	wsc:Instance	An identifier for a set of keys issued for a context. It shall be unique within the context.
createdAt	wsu:Created	This is optional element in the wsc:SecurityContextToken returned in the header.
revisedLifetime	wst:Lifetime	The revised lifetime for the SCT.
serverNonce	wst:Entropy	This contains the <i>Nonce</i> specified by the <i>Server</i> . The <i>Nonce</i> is specified with the wst:BinarySecret element. The xenc:EncryptedData element is not used in OPC UA because the <i>Message</i> body shall be encrypted.

The lifetime specifies the UTC expiration time for the security context token. The *Client* shall renew the SCT before that time by sending the RSTR/SCT *Message* again. The exact behaviour is described in IEC 62541-4.

### 6.6.5 Using the SCT

Once the *Client* receives the RSTR/SCT *Message* it can use the SCT to secure all other *Messages*.

An identifier for the SCT used shall be passed as an wsc:SecurityContextToken in each request *Message*. The response *Message* shall reference the *SecurityContextToken* used in the request.

If encryption is used it shall be applied before the signature is calculated.

Any *Message* secured with the *SecurityContextToken* shall have the following additional tokens:

- a) A wsc:DerivedKeyToken which is used to sign the body, the WS Addressing headers and the wsu:Timestamp header using the *SymmetricSignatureAlgorithm*. This key is derived from the *SecurityContextToken*. The wsc:DerivedKeyToken shall also specify a *Nonce*.
- b) A wsc:DerivedKeyToken which is used to encrypt the body of the *Message* using the *SymmetricEncryptionAlgorithm*. This key is derived from the *SecurityContextToken*. The wsc:DerivedKeyToken shall also specify a *Nonce*.

This *Message* shall have the wsa:Action and wsa:RelatesTo headers defined by WS Addressing. The *Message* shall also have a wsu:Timestamp header defined by WS Security.

### 6.6.6 Cancelling Security contexts

The *Cancel Message* defined by WS Trust implements the abstract *CloseSecureChannel* request *Message* defined in IEC 62541-4.

This *Message* shall be secured with the SCT.

## 6.7 OPC UA Secure Conversation

### 6.7.1 Overview

OPC UA Secure Conversation (UASC) is a binary version of WS-Secure Conversation. It allows secure communication over transports that do not use SOAP or XML.

UASC is designed to operate with different *TransportProtocols* that may have limited buffer sizes. For this reason, OPC UA Secure Conversation will break OPC UA *Messages* into several pieces (called '*MessageChunks*') that are smaller than the buffer size allowed by the *TransportProtocol*. UASC requires a *TransportProtocol* buffer size that is at least 8196 bytes.

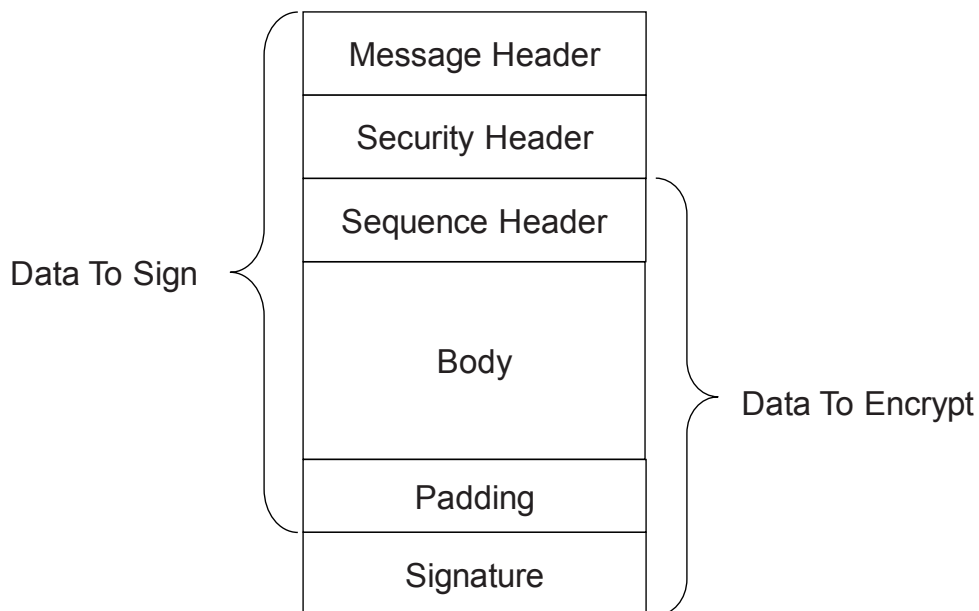
All security is applied to individual *MessageChunks* and not the entire OPC UA *Message*. A *Stack* that implements UASC is responsible for verifying the security on each *MessageChunk* received and reconstructing the original OPC UA *Message*.

All *MessageChunks* will have a 4-byte sequence assigned to them. These sequence numbers are used to detect and prevent replay attacks.

UASC requires a *TransportProtocol* that will preserve the order of *MessageChunks*, however, a UASC implementation does not necessarily process the *Messages* in the order that they were received.

### 6.7.2 MessageChunk structure

Figure 13 shows the structure of a *MessageChunk* and how security is applied to the *Message*.



IEC

**Figure 13 – OPC UA Secure Conversation MessageChunk**

Every *MessageChunk* has a *Message* header with the fields defined in Table 29.



**Table 29 – OPC UA Secure Conversation Message header**

Name	Data Type	Description
MessageType	Byte[3]	A three byte ASCII code that identifies the <i>Message</i> type. The following values are defined at this time: MSG A <i>Message</i> secured with the keys associated with a channel. OPN OpenSecureChannel <i>Message</i> . CLO CloseSecureChannel <i>Message</i> .
IsFinal	Byte	A one byte ASCII code that indicates whether the <i>MessageChunk</i> is the final chunk in a <i>Message</i> . The following values are defined at this time: C An intermediate chunk. F The final chunk. A The final chunk (used when an error occurred and the <i>Message</i> is aborted).
MessageSize	UInt32	The length of the <i>MessageChunk</i> , in bytes. This value includes size of the <i>Message</i> header.
SecureChannelId	UInt32	A unique identifier for the <i>SecureChannel</i> assigned by the <i>Server</i> . If a <i>Server</i> receives a <i>SecureChannelId</i> which it does not recognize it shall return an appropriate transport layer error. When a <i>Server</i> starts the first <i>SecureChannelId</i> used should be a value that is likely to be unique after each restart. This ensures that a <i>Server</i> restart does not cause previously connected <i>Clients</i> to accidentally 'reuse' <i>SecureChannels</i> that did not belong to them.

The *Message* header is followed by a security header which specifies what cryptography operations have been applied to the *Message*. There are two versions of the security header which depend on the type of security applied to the *Message*. The security header used for asymmetric algorithms is defined in Table 30. Asymmetric algorithms are used to secure the *OpenSecureChannel Messages*. PKCS #1 defines a set of asymmetric algorithms that may be used by UASC implementations. The *AsymmetricKeyWrapAlgorithm* element of the *SecurityPolicy* structure defined in Table 22 is not used by UASC implementations.

**Table 30 – Asymmetric algorithm Security header**

Name	Data Type	Description
SecurityPolicyUriLength	Int32	The length of the <i>SecurityPolicyUri</i> in bytes. This value shall not exceed 255 bytes.
SecurityPolicyUri	Byte[*]	The URI of the <i>Security Policy</i> used to secure the <i>Message</i> . This field is encoded as a UTF8 string without a null terminator.
SenderCertificateLength	Int32	The length of the <i>SenderCertificate</i> in bytes. This value shall not exceed <i>MaxCertificateSize</i> bytes.
SenderCertificate	Byte[*]	The X509v3 <i>Certificate</i> assigned to the sending <i>Application Instance</i> . This is a DER encoded blob. The structure of an X509 <i>Certificate</i> is defined in X509. The DER format for a <i>Certificate</i> is defined in X690 This indicates what <i>Private Key</i> was used to sign the <i>MessageChunk</i> . The <i>Stack</i> shall close the channel and report an error to the <i>Application</i> if the <i>SenderCertificate</i> is too large for the buffer size supported by the transport layer. This field shall be null if the <i>Message</i> is not signed. If the <i>Certificate</i> is signed by a CA the DER encoded CA <i>Certificate</i> may be appended after the <i>Certificate</i> in the byte array. If the CA <i>Certificate</i> is also signed by another CA this process is repeated until the entire <i>Certificate</i> chain is in the buffer or if <i>MaxSenderCertificateSize</i> limit is reached (the process stops after the last whole <i>Certificate</i> that can be added without exceeding the <i>MaxSenderCertificateSize</i> limit). Receivers can extract the <i>Certificates</i> from the byte array by using the <i>Certificate</i> size contained in DER header (see X509). Receivers that do not handle <i>Certificate</i> chains shall ignore the extra bytes.
ReceiverCertificateThumbprintLength	Int32	The length of the <i>ReceiverCertificateThumbprint</i> in bytes. The length of this field is always 20 bytes.
ReceiverCertificateThumbprint	Byte[*]	The thumbprint of the X509v3 <i>Certificate</i> assigned to the receiving <i>Application Instance</i> . The thumbprint is the SHA1 digest of the DER encoded form of the <i>Certificate</i> . This indicates what public key was used to encrypt the <i>MessageChunk</i> . This field shall be null if the <i>Message</i> is not encrypted.

The receiver shall close the communication channel if any of the fields in the security header have invalid lengths.

The *SenderCertificate*, including any chains, shall be small enough to fit into a single *MessageChunk* and leave room for at least one byte of body information. The maximum size for the *SenderCertificate* can be calculated with this formula:

```

MaxSenderCertificateSize =
  MessageChunkSize -
  12 -                // Header size
  4 -                // SecurityPolicyUriLength
  SecurityPolicyUri - // UTF-8 encoded string
  4 -                // SenderCertificateLength
  4 -                // ReceiverCertificateThumbprintLength
  20 -               // ReceiverCertificateThumbprint
  8 -                // SequenceHeader size
  1 -                // Minimum body size
  1 -                // PaddingSize if present
  Padding -         // Padding if present
  ExtraPadding -   // ExtraPadding if present
  AsymmetricSignatureSize // If present

```

The *MessageChunkSize* depends on the transport protocol but shall be at least 8196 bytes. The *AsymmetricSignatureSize* depends on the number of bits in the public key for the *SenderCertificate*. The *Int32FieldLength* is the length of an encoded Int32 value and it is always 4 bytes.

The security header used for symmetric algorithms defined in Table 31. Symmetric algorithms are used to secure all *Messages* other than the *OpenSecureChannel Messages*. FIPS 197 define symmetric encryption algorithms that UASC implementations may use. FIPS 180-2 and HMAC define some symmetric signature algorithms.

**Table 31 – Symmetric algorithm Security header**

Name	Data Type	Description
TokenId	UInt32	A unique identifier for the <i>SecureChannel SecurityToken</i> used to secure the <i>Message</i> . This identifier is returned by the <i>Server</i> in an <i>OpenSecureChannel</i> response <i>Message</i> . If a <i>Server</i> receives a <i>TokenId</i> which it does not recognize it shall return an appropriate transport layer error.

The security header is always followed by the sequence header which is defined in Table 32. The sequence header ensures that the first encrypted block of every *Message* sent over a channel will start with different data.

**Table 32 – Sequence header**

Name	Data Type	Description
SequenceNumber	UInt32	A monotonically increasing sequence number assigned by the sender to each <i>MessageChunk</i> sent over the <i>SecureChannel</i> .
RequestId	UInt32	An identifier assigned by the <i>Client</i> to OPC UA request <i>Message</i> . All <i>MessageChunks</i> for the request and the associated response use the same identifier.

*SequenceNumbers* may not be reused for any *TokenId*. The *SecurityToken* lifetime should be short enough to ensure that this never happens; however, if it does the receiver should treat it as a transport error and force a reconnect.

The *SequenceNumber* shall also monotonically increase for all *Messages* and shall not wrap around until it is greater than 4 294 966 271 (UInt32.MaxValue – 1 024). The first number after the wrap around shall be less than 1 024. Note that this requirement means that *SequenceNumbers* do not reset when a new *TokenId* is issued. The *SequenceNumber* shall be incremented by exactly one for each *MessageChunk* sent unless the communication channel was interrupted and re-established. Gaps are permitted between the

*SequenceNumber* for the last *MessageChunk* received before the interruption and the *SequenceNumber* for first *MessageChunk* received after communication was reestablished. Note that the first *MessageChunk* after a network interruption is always an *OpenSecureChannel* request or response.

The sequence header is followed by the *Message* body which is encoded with the OPC UA Binary encoding as described in 5.2.6. The body may be split across multiple *MessageChunks*.

Each *MessageChunk* also has a footer with the fields defined in Table 33.

**Table 33 – OPC UA Secure Conversation Message footer**

Name	Data Type	Description
PaddingSize	Byte	The number of padding bytes (not including the byte for the PaddingSize).
Padding	Byte[*]	Padding added to the end of the <i>Message</i> to ensure length of the data to encrypt is an integer multiple of the encryption block size. The value of each byte of the padding is equal to PaddingSize.
ExtraPaddingSize	Byte	The most significant byte of a two byte integer used to specify the padding size when the key used to encrypt the message chunk is larger than 2048 bits. This field is omitted if the key length is less than or equal to 2048 bits.
Signature	Byte[*]	The signature for the <i>MessageChunk</i> . The signature includes the all headers, all <i>Message</i> data, the PaddingSize and the Padding.

The formula to calculate the amount of padding depends on the amount of data that needs to be sent (called *BytesToWrite*). The sender shall first calculate the maximum amount of space available in the *MessageChunk* (called *MaxBodySize*) using the following formula:

$$\text{MaxBodySize} = \text{PlainTextBlockSize} * \text{Floor}((\text{MessageChunkSize} - \text{HeaderSize} - \text{SignatureSize} - 1) / \text{CipherTextBlockSize}) - \text{SequenceHeaderSize};$$

The *HeaderSize* includes the *MessageHeader* and the *SecurityHeader*. The *SequenceHeaderSize* is always 8 bytes.

During encryption a block with a size equal to *PlainTextBlockSize* is processed to produce a block with size equal to *CipherTextBlockSize*. These values depend on the encryption algorithm and may be the same.

The OPC UA *Message* can fit into a single chunk if *BytesToWrite* is less than or equal to the *MaxBodySize*. In this case the *PaddingSize* is calculated with this formula:

$$\text{PaddingSize} = \text{PlainTextBlockSize} - ((\text{BytesToWrite} + \text{SignatureSize} + 1) \% \text{PlainTextBlockSize});$$

If the *BytesToWrite* is greater than *MaxBodySize* the sender shall write *MaxBodySize* bytes with a *PaddingSize* of 0. The remaining *BytesToWrite* – *MaxBodySize* bytes shall be sent in subsequent *MessageChunks*.

The *PaddingSize* and *Padding* fields are not present if the *MessageChunk* is not encrypted.

The *Signature* field is not present if the *MessageChunk* is not signed.

### 6.7.3 MessageChunks and error handling

*MessageChunks* are sent as they are encoded. *MessageChunks* belonging to the same *Message* shall be sent sequentially. If an error occurs creating a *MessageChunk* then the sender shall send a final *MessageChunk* to the receiver that tells the receiver that an error occurred and that it should discard the previous chunks. The sender indicates that the

*MessageChunk* contains an error by setting the *IsFinal* flag to 'A' (for Abort). Table 34 specifies the contents of the *Message* about *MessageChunk*.

**Table 34 – OPC UA Secure Conversation Message abort body**

Name	Data Type	Description
Error	UInt32	The numeric code for the error. This shall be one of the values listed in Table 41.
Reason	String	A more verbose description of the error. This string shall not be more than 4 096 characters. A <i>Client</i> shall ignore strings that are longer than this.

The receiver shall check the security on the abort *MessageChunk* before processing it. If everything is ok then the receiver shall ignore the *Message* but shall not close the *SecureChannel*. The *Client* shall report the error back to the *Application* as *StatusCode* for the request. If the *Client* is the sender then it shall report the error without waiting for a response from the *Server*.

#### 6.7.4 Establishing a SecureChannel

Most *Messages* require a *SecureChannel* to be established. A *Client* does this by sending an *OpenSecureChannel* request to the *Server*. The *Server* shall validate the *Message* and the *ClientCertificate* and return an *OpenSecureChannel* response. Some of the parameters defined for the *OpenSecureChannel* service are specified in the security header (see 6.7.2) instead of the body of the *Message*. For this reason, the *OpenSecureChannel* Service is not the same as the one specified in IEC 62541-4. Table 35 lists the parameters that appear in the body of the *Message*.

**Table 35 – OPC UA Secure Conversation OpenSecureChannel Service**

Name	Data Type
<b>Request</b>	
RequestHeader	RequestHeader
ClientProtocolVersion	UInt32
RequestType	SecurityTokenRequestType
SecurityMode	MessageSecurityMode
ClientNonce	ByteString
RequestedLifetime	Int32
<b>Response</b>	
ResponseHeader	ResponseHeader
ServerProtocolVersion	UInt32
SecurityToken	ChannelSecurityToken
SecureChannelId	UInt32
TokenId	UInt32
CreatedAt	DateTime
RevisedLifetime	Int32
ServerNonce	ByteString

The *ClientProtocolVersion* and *ServerProtocolVersion* parameters are not defined in IEC 62541-4 and are added to the *Message* to allow backward compatibility if OPC UA-SecureConversation needs to be updated in the future. Receivers always accept numbers greater than the latest version that they support. The receiver with the higher version number is expected to ensure backward compatibility.

If OPC UA-SecureConversation is used with the OPC UA-TCP protocol (see 7.1) then the version numbers specified in the *OpenSecureChannel* Messages shall be the same as the version numbers specified in the OPC UA-TCP protocol *Hello/Acknowledge* Messages. The receiver shall close the channel and report a *Bad\_ProtocolVersionUnsupported* error if there is a mismatch.

The *Server* shall return an error response as described in IEC 62541-4 if there are any errors with the parameters specified by the *Client*.

The *RevisedLifetime* tells the *Client* when it shall renew the *SecurityToken* by sending another *OpenSecureChannel* request. The *Client* shall continue to accept the old *SecurityToken* until it receives the *OpenSecureChannel* response. The *Server* has to accept requests secured with the old *SecurityToken* until that *SecurityToken* expires or until it receives a *Message* from the *Client* secured with the new *SecurityToken*. The *Server* shall reject renew requests if the *SenderCertificate* is not the same as the one used to create the *SecureChannel* or if there is a problem decrypting or verifying the signature. The *Client* shall abandon the *SecureChannel* if the *Certificate* used to sign the response is not the same as the *Certificate* used to encrypt the request.

The *OpenSecureChannel Messages* are not signed or encrypted if the *SecurityMode* is *None*. The *Nonces* are ignored and should be set to null. The *SecureChannelId* and the *TokenId* are still assigned but no security is applied to *Messages* exchanged via the channel. The *SecurityToken* shall still be renewed before the *RevisedLifetime* expires. Receivers shall still ignore invalid or expired *TokenIds*.

If the communication channel breaks the *Server* shall maintain the *Secure Channel* long enough to allow the *Client* to reconnect. The *ReviseLifetime* parameter also tells the *Client* how long the *Server* will wait. If the *Client* cannot reconnect within that period it shall assume the *SecureChannel* has been closed.

The *AuthenticationToken* in the *RequestHeader* shall be set to null.

If an error occurs after the *Server* has verified *Message* security it shall return a *ServiceFault* instead of a *OpenSecureChannel* response. The *ServiceFault Message* is described in IEC 62541-4.

If the *SecurityMode* is not *None* then the *Server* shall verify that a *SenderCertificate* and a *ReceiverCertificateThumbprint* were specified in the *SecurityHeader*.

### 6.7.5 Deriving keys

Once the *SecureChannel* is established the *Messages* are signed and encrypted with keys derived from the *Nonces* exchanged in the *OpenSecureChannel* call. These keys are derived by passing the *Nonces* to a pseudo-random function which produces a sequence of bytes from a set of inputs. A pseudo-random function is represented by the following function declaration:

```
Byte[] PRF(
    Byte[] secret,
    Byte[] seed,
    Int32 length,
    Int32 offset)
```

Where *length* is the number of bytes to return and *offset* is a number of bytes from the beginning of the sequence.

The lengths of the keys that need to be generated depend on the *SecurityPolicy* used for the channel. The following information is specified by the *SecurityPolicy*:

- a) *SigningKeyLength* (from the *DerivedSignatureKeyLength*);
- b) *EncryptingKeyLength* (implied by the *SymmetricEncryptionAlgorithm*);
- c) *EncryptingBlockSize* (implied by the *SymmetricEncryptionAlgorithm*).

The parameters passed to the pseudo random function are specified in Table 36.



**Table 36 – Cryptography key generation parameters**

Key	Secret	Seed	Length	Offset
ClientSigningKey	ServerNonce	ClientNonce	SigningKeyLength	0
ClientEncryptingKey	ServerNonce	ClientNonce	EncryptingKeyLength	SigningKeyLength
ClientInitializationVector	ServerNonce	ClientNonce	EncryptingBlockSize	SigningKeyLength+ EncryptingKeyLength
ServerSigningKey	ClientNonce	ServerNonce	SigningKeyLength	0
ServerEncryptingKey	ClientNonce	ServerNonce	EncryptingKeyLength	SigningKeyLength
ServerInitializationVector	ClientNonce	ServerNonce	EncryptingBlockSize	SigningKeyLength+ EncryptingKeyLength

The *Client* keys are used to secure *Messages* sent by the *Client*. The *Server* keys are used to secure *Messages* sent by the *Server*.

The SSL/TLS specification defines a pseudo random function called P\_SHA1 which is used for some *SecurityProfiles*. The P\_SHA1 algorithm is defined as follows:

$$P\_SHA1(secret, seed) = HMAC\_SHA1(secret, A(1) + seed) + \\ HMAC\_SHA1(secret, A(2) + seed) + \\ HMAC\_SHA1(secret, A(3) + seed) + \dots$$

Where A(n) is defined as:

$$A(0) = seed$$

$$A(n) = HMAC\_SHA1(secret, A(n-1))$$

+ indicates that the results are appended to previous results.

### 6.7.6 Verifying Message Security

The contents of the *MessageChunk* shall not be interpreted until the *Message* is decrypted and the signature and sequence number verified.

If an error occurs during *Message* verification the receiver shall close the communication channel. If the receiver is the *Server* it shall also send a transport error *Message* before closing the channel. Once the channel is closed the *Client* shall attempt to re-open the channel and request a new *SecurityToken* by sending an *OpenSecureChannel* request. The mechanism for sending transport errors to the *Client* depends on the communication channel.

The receiver shall first check the *SecureChannelId*. This value may be 0 if the *Message* is an *OpenSecureChannel* request. For other *Messages* it shall report a *Bad\_SecureChannelUnknown* error if the *SecureChannelId* is not recognized. If the *Message* is an *OpenSecureChannel* request and the *SecureChannelId* is not 0 then the *SenderCertificate* shall be the same as the *SenderCertificate* used to create the channel.

If the *Message* is secured with asymmetric algorithms then the receiver shall verify that it supports the requested *SecurityPolicy*. If the *Message* is the response sent to the *Client* then the *SecurityPolicy* shall be the same as the one specified in the request. In the *Server* the *SecurityPolicy* shall be the same as the one used to originally create the *SecureChannel*. The receiver shall check that the *Certificate* is trusted first and return *Bad\_CertificateUntrusted* on error. The receiver shall then verify the *SenderCertificate* using the rules defined in IEC 62541-4. The receiver shall report the appropriate error if *Certificate* validation fails. The receiver shall verify the *ReceiverCertificateThumbprint* and report a *Bad\_CertificateUnknown* error if it does not recognize it.

If the *Message* is secured with symmetric algorithms then a *Bad\_SecureChannelTokenUnknown* error shall be reported if the *TokenId* refers to a *SecurityToken* that has expired or is not recognized.

If decryption or signature validation fails then a *Bad\_SecurityChecksFailed* error is reported. If an implementation allows multiple *SecurityModes* to be used the receiver shall also verify that the *Message* was secured properly as required by the *SecurityMode* specified in the *OpenSecureChannel* request.

After the security validation is complete the receiver shall verify the *RequestId* and the *SequenceNumber*. If these checks fail a *Bad\_SecurityChecksFailed* error is reported. The *RequestId* only needs to be verified by the *Client* since only the *Client* knows if it is valid or not.

At this point the *SecureChannel* knows it is dealing with an authenticated *Message* that was not tampered with or resent. This means the *SecureChannel* can return secured error responses if any further problems are encountered.

*Stacks* that implement UASC shall have a mechanism to log errors when invalid *Messages* are discarded. This mechanism is intended for developers, systems integrators and administrators to debug network system configuration issues and to detect attacks on the network.

## 7 Transport Protocols

### 7.1 OPC UA TCP

#### 7.1.1 Overview

OPC UA TCP is a simple TCP based protocol that establishes a full duplex channel between a *Client* and *Server*. This protocol has two key features that differentiate it from HTTP. First, this protocol allows responses to be returned in any order. Second, this protocol allows responses to be returned on a different TCP transport end-point if communication failures cause temporary TCP session interruption.

The OPC UA TCP protocol is designed to work with the *SecureChannel* implemented by a layer higher in the stack. For this reason, the OPC UA TCP protocol defines its interactions with the *SecureChannel* in addition to the wire protocol.

#### 7.1.2 Message structure

Every OPC UA TCP *Message* has a header with the fields defined in Table 37.

**Table 37 – OPC UA TCP Message header**

Name	Type	Description
MessageType	Byte[3]	A three byte ASCII code that identifies the <i>Message</i> type. The following values are defined at this time: HEL a <i>Hello Message</i> . ACK an <i>Acknowledge Message</i> . ERR an <i>Error Message</i> . The <i>SecureChannel</i> layer defines additional values which the OPC UA TCP layer shall accept.
Reserved	Byte[1]	Ignored. shall be set to the ASCII codes for 'F' if the <i>MessageType</i> is one of the values supported by the OPC UA TCP protocol.
MessageSize	UInt32	The length of the <i>Message</i> , in bytes. This value includes the 8 bytes for the <i>Message</i> header.

The layout of the OPC UA TCP *Message* header is intentionally identical to the first 8 bytes of the OPC UA Secure Conversation *Message* header defined in Table 29. This allows the OPC UA TCP layer to extract the *SecureChannel Messages* from the incoming stream even if it does not understand their contents.

The OPC UA TCP layer shall verify the *MessageType* and make sure the *MessageSize* is less than the negotiated *ReceiveBufferSize* before passing any *Message* onto the *SecureChannel* layer.

The Hello *Message* has the additional fields shown in Table 38.

**Table 38 – OPC UA TCP Hello Message**

Name	Data Type	Description
ProtocolVersion	UInt32	The latest version of the OPC UA TCP protocol supported by the <i>Client</i> . The <i>Server</i> may reject the <i>Client</i> by returning <i>Bad_ProtocolVersionUnsupported</i> . If the <i>Server</i> accepts the connection is responsible for ensuring that it returns <i>Messages</i> that conform to this version of the protocol. The <i>Server</i> shall always accept versions greater than what it supports.
ReceiveBufferSize	UInt32	The largest <i>MessageChunk</i> that the sender can receive. This value shall be greater than 8 192 bytes.
SendBufferSize	UInt32	The largest <i>MessageChunk</i> that the sender will send. This value shall be greater than 8 192 bytes.
MaxMessageSize	UInt32	The maximum size for any response <i>Message</i> . The <i>Server</i> shall abort the <i>Message</i> with a <i>Bad_ResponseTooLarge StatusCode</i> if a response <i>Message</i> exceeds this value. The mechanism for aborting <i>Messages</i> is described fully in 6.7.3. The <i>Message</i> size is calculated using the unencrypted <i>Message</i> body. A value of zero indicates that the <i>Client</i> has no limit.
MaxChunkCount	UInt32	The maximum number of chunks in any response <i>Message</i> . The <i>Server</i> shall abort the <i>Message</i> with a <i>Bad_ResponseTooLarge StatusCode</i> if a response <i>Message</i> exceeds this value. The mechanism for aborting <i>Messages</i> is described fully in 6.7.3. A value of zero indicates that the <i>Client</i> has no limit.
EndpointUrl	String	The URL of the <i>Endpoint</i> which the <i>Client</i> wished to connect to. The encoded value shall be less than 4 096 bytes. <i>Servers</i> shall return a <i>Bad_TcpUrlRejected</i> error and close the connection if the length exceeds 4 096 or if it does not recognize the resource identified by the URL.

The *EndpointUrl* parameter is used to allow multiple *Servers* to share the same port on a machine. The process listening (also known as the proxy) on the port would connect to the *Server* identified by the *EndpointUrl* and would forward all *Messages* to the *Server* via this socket. If one socket closes then the proxy shall close the other socket.

The *Acknowledge Message* has the additional fields shown in Table 39.

**Table 39 – OPC UA TCP Acknowledge Message**

Name	Type	Description
ProtocolVersion	UInt32	The latest version of the OPC UA TCP protocol supported by the <i>Server</i> . If the <i>Client</i> accepts the connection is responsible for ensuring that it sends <i>Messages</i> that conform to this version of the protocol. The <i>Client</i> shall always accept versions greater than what it supports.
ReceiveBufferSize	UInt32	The largest <i>MessageChunk</i> that the sender can receive. This value shall not be larger than what the <i>Client</i> requested in the Hello <i>Message</i> . This value shall be greater than 8 192 bytes.
SendBufferSize	UInt32	The largest <i>MessageChunk</i> that the sender will send. This value shall not be larger than what the <i>Client</i> requested in the Hello <i>Message</i> . This value shall be greater than 8 192 bytes.
MaxMessageSize	UInt32	The maximum size for any request <i>Message</i> . The <i>Client</i> shall abort the <i>Message</i> with a <i>Bad_RequestTooLarge StatusCode</i> if a request <i>Message</i> exceeds this value. The mechanism for aborting <i>Messages</i> is described fully in 6.7.3. The <i>Message</i> size is calculated using the unencrypted <i>Message</i> body. A value of zero indicates that the <i>Server</i> has no limit.
MaxChunkCount	UInt32	The maximum number of chunks in any request <i>Message</i> . The <i>Client</i> shall abort the <i>Message</i> with a <i>Bad_RequestTooLarge StatusCode</i> if a request <i>Message</i> exceeds this value. The mechanism for aborting <i>Messages</i> is described fully in 6.7.3. A value of zero indicates that the <i>Server</i> has no limit.

The *Error Message* has the additional fields shown in Table 40.

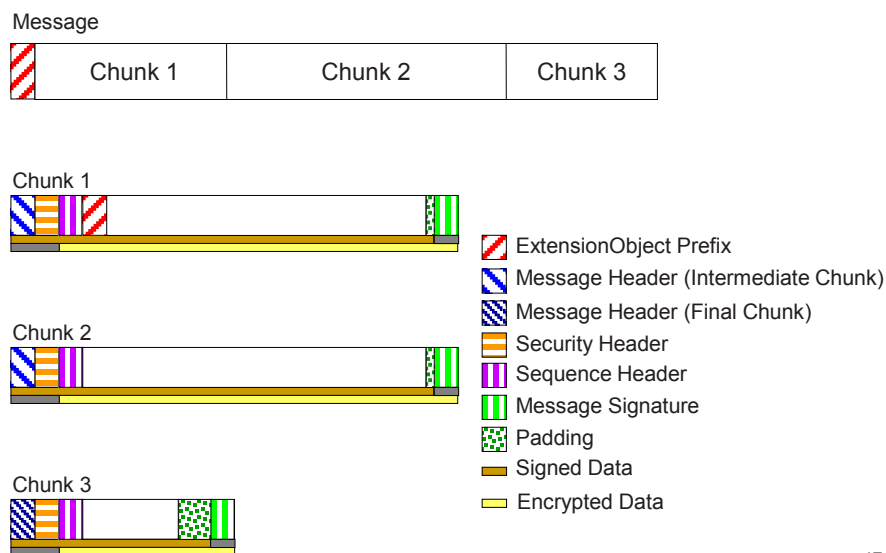


**Table 40 – OPC UA TCP Error Message**

Name	Type	Description
Error	UInt32	The numeric code for the error. This shall be one of the values listed in Table 41.
Reason	String	A more verbose description of the error. This string shall not be more than 4 096 characters. A <i>Client</i> shall ignore strings that are longer than this.

Figure 14 illustrates the structure of a *Message* placed on the wire. This includes also illustrates how the *Message* elements defined by the OPC UA Binary Encoding mapping (see 5.2) and the OPC UA Secure Conversation mapping (see 6.7) relate to the OPC UA TCP *Messages*.

The socket is always closed gracefully by the *Server* after it sends an *Error Message*.



IEC

**Figure 14 – OPC UA TCP Message structure**

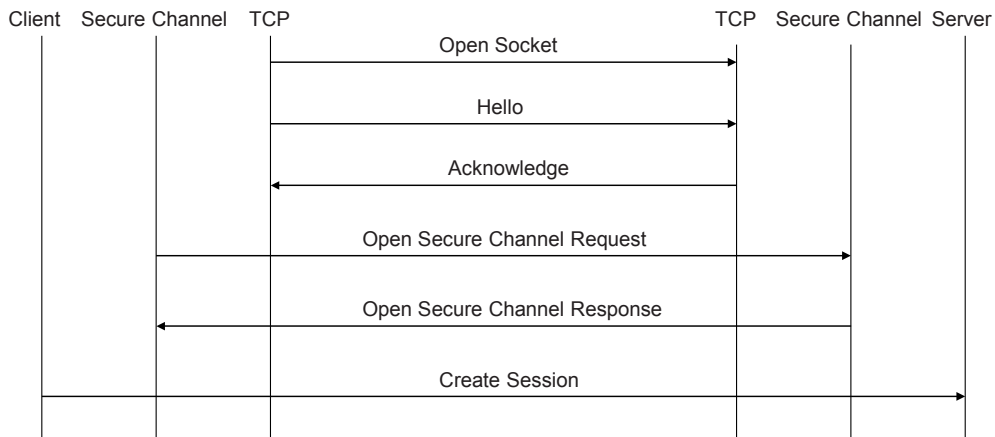
### 7.1.3 Establishing a connection

Connections are always initiated by the *Client* which creates the socket before it sends the first *OpenSecureChannel* request. After creating the socket the first *Message* sent shall be a *Hello* which specifies the buffer sizes that the *Client* supports. The *Server* shall respond with an *Acknowledge Message* which completes the buffer negotiation. The negotiated buffer size shall be reported to the *SecureChannel* layer. The negotiated *SendBufferSize* specifies the size of the *MessageChunks* to use for *Messages* sent over the connection.

The *Hello/Acknowledge Messages* may only be sent once. If they are received again the receiver shall report an error and close the socket. *Servers* shall close any socket after a period of time if it does not receive a *Hello Message*. This period of time shall be configurable and have a default value which does not exceed two minutes.

The *Client* sends the *OpenSecureChannel* request once it receives the *Acknowledge* back from the *Server*. If the *Server* accepts the new channel it shall associate the socket with the *SecureChannelId*. The *Server* uses this association to determine which socket to use when it has to send a response to the *Client*. The *Client* does the same when it receives the *OpenSecureChannel* response.

The sequence of *Messages* when establishing a OPC UA TCP connection are shown in Figure 15.



IEC

**Figure 15 – Establishing a OPC UA TCP connection**

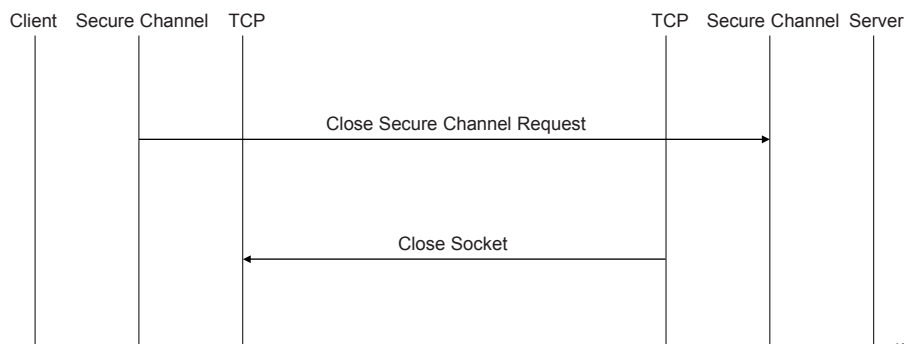
The *Server Application* does not do any processing while the *SecureChannel* is negotiated; however, the *Server Application* shall to provide the *Stack* with the list of trusted *Certificates*. The *Stack* shall provide notifications to the *Server Application* whenever it receives an *OpenSecureChannel* request. These notifications shall include the *OpenSecureChannel* or *Error* response returned to the *Client*.

**7.1.4 Closing a connection**

The *Client* closes the connection by sending a *CloseSecureChannel* request and closing the socket gracefully. When the *Server* receives this *Message* it shall release all resources allocated for the channel. The *Server* does not send a *CloseSecureChannel* response.

If security verification fails for the *CloseSecureChannel Message* then the *Server* shall report the error and close the socket. The *Server* shall allow the *Client* to attempt to reconnect.

The sequence of *Messages* when closing an OPC UA TCP connection is shown in Figure 16.



IEC

**Figure 16 – Closing a OPC UA TCP connection**

The *Server Application* does not do any processing when the *SecureChannel* is closed; however, the *Stack* shall provide notifications to the *Server Application* whenever a *CloseSecureChannel* request is received or when the *Stack* cleans up an abandoned *SecureChannel*.

### 7.1.5 Error handling

When a fatal error occurs the *Server* shall send an *Error Message* to the *Client* and close the socket. When a *Client* encounters one of these errors, it shall also close the socket but does not send an *Error Message*. After the socket is closed a *Client* shall try to reconnect automatically using the mechanisms described in 7.1.6.

The possible OPC UA TCP errors are defined in Table 41.

**Table 41 – OPC UA TCP error codes**

Name	Description
TcpServerTooBusy	The <i>Server</i> cannot process the request because it is too busy. It is up to the <i>Server</i> to determine when it needs to return this <i>Message</i> . A <i>Server</i> can control the how frequently a <i>Client</i> reconnects by waiting to return this error.
TcpMessageTypeInvalid	The type of the <i>Message</i> specified in the header invalid. Each <i>Message</i> starts with a 4 byte sequence of ASCII values that identifies the <i>Message</i> type. The <i>Server</i> returns this error if the <i>Message</i> type is not accepted. Some of the <i>Message</i> types are defined by the <i>SecureChannel</i> layer.
TcpSecureChannelUnknown	The <i>SecureChannelId</i> and/or <i>TokenId</i> are not currently in use. This error is reported by the <i>SecureChannel</i> layer.
TcpMessageTooLarge	The size of the <i>Message</i> specified in the header is too large. The <i>Server</i> returns this error if the <i>Message</i> size exceeds its maximum buffer size or the receive buffer size negotiated during the <i>Hello/Acknowledge</i> exchange.
TcpTimeout	A timeout occurred while accessing a resource. It is up to the <i>Server</i> to determine when a timeout occurs.
TcpNotEnoughResources	There are not enough resources to process the request. The <i>Server</i> returns this error when it runs out of memory or encounters similar resource problems. A <i>Server</i> can control the how frequently a <i>Client</i> reconnects by waiting to return this error.
TcpInternalError	An internal error occurred. This should only be returned if an unexpected configuration or programming error occurs.
TcpUrlRejected	The <i>Server</i> does not recognize the <i>EndpointUrl</i> specified.
SecurityChecksFailed	The <i>Message</i> was rejected because it could not be verified.
RequestInterrupted	The request could not be sent because of a network interruption.
RequestTimeout	Timeout occurred while processing the request.
SecureChannelClosed	The secure channel has been closed.
SecureChannelTokenUnknown	The <i>SecurityToken</i> has expired or is not recognized.
CertificateUntrusted	The sender <i>Certificate</i> is not trusted by the receiver.
CertificateTimelInvalid	The sender <i>Certificate</i> has expired or is not yet valid.
CertificateIssuerTimelInvalid	The issuer for the sender <i>Certificate</i> has expired or is not yet valid.
CertificateUseNotAllowed	The sender's <i>Certificate</i> may not be used for establishing a secure channel.
CertificateIssuerUseNotAllowed	The issuer <i>Certificate</i> may not be used as a <i>Certificate Authority</i> .
CertificateRevocationUnknown	Could not verify the revocation status of the sender's <i>Certificate</i> .
CertificateIssuerRevocationUnknown	Could not verify the revocation status of the issuer <i>Certificate</i> .
CertificateRevoked	The sender <i>Certificate</i> has been revoked by the issuer.
IssuerCertificateRevoked	The issuer <i>Certificate</i> has been revoked by its issuer.
CertificateUnknown	The receiver <i>Certificate</i> thumbprint is not recognized by the receiver.

The numeric values for these error codes are defined in A.2.

### 7.1.6 Error recovery

Once the *SecureChannel* has been established, the *Client* shall go into an error recovery state whenever the socket breaks or if the *Server* returns an OPC UA TCP *Error Message* as defined in Table 40. While in this state the *Client* periodically attempts to reconnect to the *Server*. If the reconnect succeeds the *Client* sends a *Hello* followed by an *OpenSecureChannel* request (see 6.7.4) that re-authenticates the *Client* and associates the new socket with the existing *SecureChannel*.

The *Client* shall wait between reconnect attempts. The first reconnect shall happen immediately. After that, the wait period should start as 1 second and increase gradually to a maximum of 2 minutes. One sequence would double the period each attempt until reaching

the maximum. In other words, the *Client* would use the following wait periods: { 0, 1, 2, 4, 8, 16, 32, 64, 120, 120, ...}. The *Client* shall keep attempting to reconnect until the *SecureChannel* is closed or after the period equal to the *RevisedLifetime* of the last *SecurityToken* elapses.

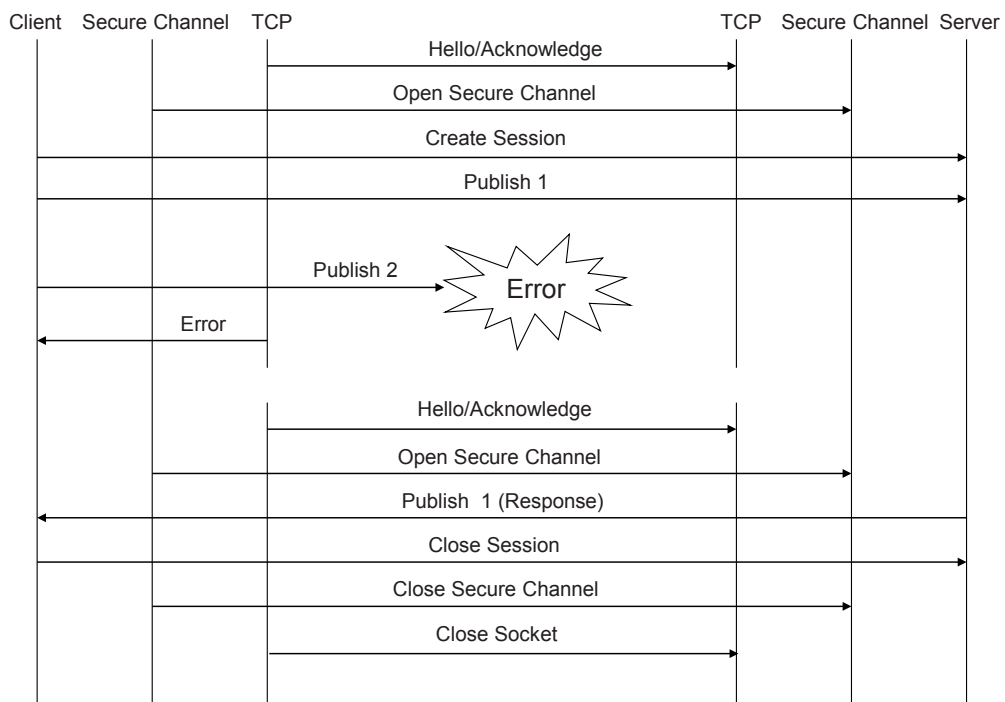
The *Stack* in the *Server* should not discard responses if there is no connection immediately available. It should wait and see if the *Client* creates a new socket. It is up to the *Server* stack implementation to decide how long it will wait and how many responses it is willing to hold onto.

The *Stack* in the *Client* shall not fail requests that have already been sent and are waiting for a response when the socket is closed. However, these requests may timeout and report a *Bad\_TcpRequestTimeout* error to the *Application*. If the *Client* sends a new request the stack shall either buffer the request or return a *Bad\_TcpRequestInterrupted* error. The *Client* can stop the reconnect process by closing the *SecureChannel*.

The *Server* may abandon the *SecureChannel* before a *Client* is able to reconnect. If this happens the *Client* will get a *Bad\_TcpSecureChannelUnknown* error in response to the *OpenSecureChannel* request. The *Stack* shall return this error to the *Application* that can attempt to create a new *SecureChannel*.

The negotiated buffer sizes should never change when a connection is recovered; however, the buffer sizes are negotiated before the *Server* knows whether the socket is being used for an existing *SecureChannel* or a new one. A *Client* shall treat this as a fatal error, close the *SecureChannel* and returns an *Bad\_TcpSecureChannelClosed* error to the *Application*.

The sequence of *Messages* when recovering an OPC UA TCP connection is shown in Figure 17.



IEC

Figure 17 – Recovering an OPC UA TCP connection

## 7.2 SOAP/HTTP

### 7.2.1 Overview

SOAP describes an XML based syntax for exchanging *Messages* between *Applications*. OPC UA *Messages* are exchanged using SOAP by serializing the OPC UA *Messages* using one of the supported encodings described in Clause 5 and inserting that encoded *Message* into the body of the SOAP *Message*.

All OPC UA *Applications* that support the SOAP/HTTP transport shall support SOAP 1.2 as described in SOAP Part 1.

All OPC UA *Messages* are exchanged using the request-response *Message* exchange pattern defined in SOAP Part 2 even if the OPC UA service does not specify any output parameters. In these cases, the *Server* shall return an empty response *Message* that tells the *Client* that no errors occurred.

WS-I Basic Profile 1.1 defines best practices when using SOAP *Messages* which will help ensure interoperability. All OPC UA implementations shall conform to this specification.

HTTP is the network communication protocol used to exchange SOAP *Messages*. An OPC UA service request *Message* is always sent by the *Client* in the body of an HTTP POST request. The *Server* returns an OPC UA response *Message* in the body of the HTTP response. The HTTP binding for SOAP is described completely in SOAP Part 2.

OPC UA does not define any SOAP headers; however, SOAP *Messages* containing OPC UA *Messages* will include headers used by the other WS specifications in the stack.

SOAP faults are returned only for errors that occurred with in the SOAP stack. Errors that occur within in the *Application* are returned as OPC UA error response *Messages* as described in IEC 62541-4.

WS Addressing defines standard headers used to route SOAP *Messages* through intermediaries. Implementations shall support the WS-Addressing headers listed Table 42.

**Table 42 – WS-Addressing headers**

Header	Request	Response
wsa:To	Required	Optional
wsa:From	Optional	Optional
wsa:ReplyTo	Required	Not Used
wsa:Action	Required	Required
wsa:MessageID	Required	Optional
wsa:RelatesTo	Not Used	Required

Note that WS-Addressing defines standard URIs to use in the ReplyTo and From headers when a *Client* does not have an externally accessible endpoint. In these cases, the SOAP response is always returned to the *Client* using the same communication channel that sent the request.

OPC UA *Servers* shall ignore the wsa:FaultTo header if it is specified in a request.

### 7.2.2 XML Encoding

The OPC UA XML Encoding specifies a way to represent an OPC UA *Message* as an XML element. This element is added to the SOAP *Message* as the only child of the SOAP body element.

If an error occurs in the *Server* while parsing the request body, the *Server* may return a SOAP fault or it may return an OPC UA error response.

The SOAP Action associated with an XML encoded request *Message* always has the form:

```
http://opcfoundation.org/UA/2008/02/Services.wsdl/<service name>
```

Where <service name> is the name of the OPC UA *Service* being invoked.

The SOAP Action associated with an XML encoded response *Message* always has the form:

```
http://opcfoundation.org/UA/2008/02/Services.wsdl/<service name>Response
```

### 7.2.3 OPC UA Binary Encoding

The OPC UA Binary Encoding specifies a way to represent an OPC UA *Message* as a sequence of bytes. These bytes sequences shall be encoded in the SOAP body using the Base64 data format.

The Base64 data format is a UTF-7 representation of binary data that is less efficient than raw binary data, however, many OPC UA *Applications* that exchange *Messages* using SOAP will find that encoding OPC UA *Messages* in OPC UA Binary and then encoding the binary in Base64 is more efficient than encoding everything in XML.

The WSDL definition for a OPC UA Binary encoded request *Message* is:

```
<xs:element name="InvokeServiceRequest" type="xs:base64Binary" nillable="true" />  
  
<wsdl:message name="InvokeServiceMessage">  
  <wsdl:part name="input" element="s0:InvokeServiceRequest"/>  
</wsdl:message>
```

The SOAP Action associated with an OPC UA Binary encoded request *Message* always has the form:

```
http://opcfoundation.org/UA/2008/02/Services.wsdl/InvokeService
```

The WSDL definition for an OPC UA Binary encoded response *Message* is:

```
<xs:element name="InvokeServiceResponse" type="xs:base64Binary" nillable="true" />  
  
<wsdl:message name="InvokeServiceResponseMessage">  
  <wsdl:part name="output" element="s0:InvokeServiceResponse"/>  
</wsdl:message>
```

The SOAP Action associated with an OPC UA Binary encoded response *Message* always has the form:

```
http://opcfoundation.org/UA/2008/02/Services.wsdl/ InvokeServiceResponse
```

## 7.3 HTTPS

### 7.3.1 Overview

HTTPS refers HTTP *Messages* exchanged over a SSL/TLS connection. The syntax of the HTTP *Messages* does not change and the only difference is a TLS connection is created instead of a TCP/IP connection. This implies this that profiles which use this transport can also be used with HTTP when security is not a concern.

HTTPS is a protocol that provides transport security. This means all bytes are secured as they are sent without considering the *Message* boundaries. Transport security can only work for point to point communication and does not allow untrusted intermediaries or proxy servers to handle traffic.

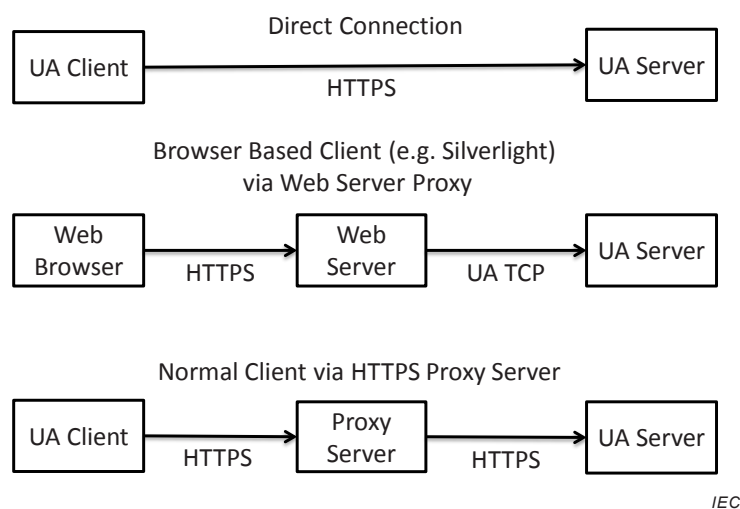
The *SecurityPolicy* shall be specified, however, it only affects the algorithms used for signing the *Nonces* during the *CreateSession/ActivateSession* handshake. A *SecurityPolicy* of *None* indicates that the *Nonces* do not need to be signed. The *SecurityMode* is set to *Sign* unless the *SecurityPolicy* is *None*; in this case the *SecurityMode* shall be set to *None*. If a *UserIdentityToken* is to be encrypted it shall be explicitly specified in the *UserTokenPolicy*.

An HTTP Header called 'OPCUA-SecurityPolicy' is used by the *Client* to tell the *Server* what *SecurityPolicy* it is using if there are multiple choices available. The value of the header is the URI for the *SecurityPolicy*. If the *Client* omits the header then the *Server* shall assume a *SecurityPolicy* of *None*.

All HTTPS communications via a URL shall be treated as a single *SecureChannel* that is shared by multiple *Clients*. *Stacks* shall provide a unique identifier for the *SecureChannel* which allows *Applications* correlate a request with a *SecureChannel*. This means that *Sessions* can only be considered secure if the *AuthenticationToken* (see IEC 62541-4) is long (>20 bytes) and HTTPS encryption is enabled.

The cryptography algorithms used by HTTPS have no relationship to the *EndpointDescription SecurityPolicy* and are determined by the policies set for HTTPS and are outside the scope of OPC UA.

Figure 18 illustrates a few scenarios where the HTTPS transport could be used.



**Figure 18 – Scenarios for the HTTPS Transport**

In some scenarios, HTTPS communication will rely on an intermediary which is not trusted by the applications. If this is the case then the HTTPS transport cannot be used to ensure security and the applications will have to establish a secure tunnel like a VPN before attempting any OPC UA related communication.

Applications which support the HTTPS transport shall support HTTP 1.1 and SSL/TLS 1.0.

Some HTTPS implementations require that all *Servers* have a *Certificate* with a *Common Name* (CN) that matches the DNS name of the *Server* machine. This means that a *Server* with multiple DNS names will need multiple HTTPS certificates. If multiple *Servers* are on the



same machine they may share HTTPS certificates. This means that *ApplicationCertificates* are not the same as *HTTPS Certificates*. *Applications* which use the HTTPS transport and require *Application* authentication shall check *Application Certificates* during the CreateSession/ActivateSession handshake.

*HTTPS Certificates* can be automatically generated; however, this will cause problems for *Clients* operating inside a restricted environment such as a web browser. Therefore, *HTTPS* certificates should be issued by an authority which is accepted by all web browsers which need to access the *Server*. The set of *Certificate* authorities accepted by the web browsers is determined by the organization that manages the *Client* machines. *Client* applications that are not running inside a web may use the trust list that is used for *Application Certificates*.

*HTTPS* connections have an unpredictable lifetime. Therefore, *Servers* must rely on the *AuthenticationToken* passed in the *RequestHeader* to determine the identity of the *Client*. This means the *AuthenticationToken* shall be a randomly generated value with at least 32 bytes of data and *HTTPS* with signing and encryption shall always be used.

*HTTPS* allows *Clients* to have certificates; however, they are not required by the *HTTPS* transport. A *Server* shall allow *Clients* to connect without an *HTTPS Certificate*.

*HTTP 1.1* supports *Message* chunking where the *Content-Length* header in the request response is set to “chunked” and each chunk is prefixed by its size in bytes. All applications that support the *HTTPS* transport shall supporting *HTTP* chunking.

### 7.3.2 XML Encoding

This *TransportProfile* implements the *OPC UA Services* using a *SOAP* request-response message pattern over an *HTTPS* connection.

The body of the *HTTP Messages* shall be a *SOAP 1.2 Message* (see *SOAP Part 1*). *WS-Addressing* headers are optional. The contents of the *SOAP* body and the *SOAP* action are the same as specified in 7.2.2 and 7.2.3.

All requests shall be *HTTP POST* requests. The *Content-type* shall be “application/soap+xml” and the *charset* and *action* parameters shall be specified. The *charset* parameter shall be “utf-8” and the *action* parameter shall be the *URI* for the *SOAP* action.

An example *HTTP* request header is:

```
POST /UA/SampleServer HTTP/1.1
Content-Type: application/soap+xml; charset="utf-8";
    action="http://opcfoundation.org/UA/2008/02/Services.wsdl/Read"
Content-Length: nnnn
```

The *action* parameter appears on the same line as the *Content-Type* declaration.

An example request *Message* (see 7.2.3):

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Body>
    <ReadRequest xmlns="http://opcfoundation.org/UA/2008/02/Types.xsd">
      ...
    </ReadRequest>
  </s:Body>
</s:Envelope>
```

An example *HTTP* response header is:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8";
```



```

    action="http://opcfoundation.org/UA/2008/02/Services.wsdl/ReadResponse"
Content-Length: nnnn

```

The action parameter appears on the same line as the Content-Type declaration.

An example response *Message*:

```

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Body>
    <ReadResponse xmlns="http://opcfoundation.org/UA/2008/02/Types.xsd">
      ...
    </ReadResponse>
  </s:Body>
</s:Envelope>

```

### 7.3.3 OPC UA Binary Encoding

This *TransportProfile* implements the OPC UA *Services* using an OPC UA Binary encoded *Messages* exchanged over an HTTPS connection.

Applications which support the HTTPS *Profile* shall support HTTP 1.1.

The body of the HTTP *Messages* shall be OPC UA Binary encoded blob. The Content-type shall be "application/octet-stream".

An example HTTP request header is:

```

POST /UA/SampleServer HTTP/1.1
Content-Type: application/octet-stream;
Content-Length: nnnn

```

An example HTTP response header is:

```

HTTP/1.1 200 OK
Content-Type: application/octet-stream;
Content-Length: nnnn

```

The *Message* body is the request or response structure encoded as an *ExtensionObject* in OPC UA Binary.

## 7.4 Well known addresses

The *Local Discovery Server* (LDS) is an OPC UA *Server* that implements the *Discovery Service Set* defined in IEC 62541-4. If an LDS is installed on a machine it shall use one or more of the well-known addresses defined in Table 43.

**Table 43 – Well known addresses for Local Discovery Servers**

Transport Mapping	URL	Notes
SOAP/HTTP	http://localhost/UADiscovery	May require integration with a web <i>Server</i> like IIS.
SOAP/HTTP	http://localhost:52601/UADiscovery	Alternate if it Port 80 cannot be used by the LDS.
OPC UA TCP	opc.tcp://localhost:4840/UADiscovery	
OPC UA HTTPS	https://localhost:4843/UADiscovery	

OPC UA *Applications* that make use of the LDS shall allow administrators to change the well known addresses used within a system.

The *Endpoint* used by *Servers* to register with the LDS shall be the base address with the path "/registration" appended to it (e.g. <http://localhost/UADiscovery/registration>). OPC UA *Servers* shall allow administrators to configure the address to use for registration.

Each OPC UA *Server Application* implements the *Discovery Service Set*. If the OPC UA *Server* requires a different address for this *Endpoint* it shall create the address by appending the path “/discovery” to its base address.

## 8 Normative Contracts

### 8.1 OPC Binary Schema

The normative contract for the OPC UA Binary encoded *Messages* is an OPC Binary Schema. This file defines the structure of all types and *Messages*. The syntax for an OPC Binary Type Schema is described in IEC 62541-3. This schema captures normative names for types and their fields as well the order the fields appear when encoded. The data type of each field is also captured.

### 8.2 XML Schema and WSDL

The normative contract for the OPC UA XML encoded *Messages* is an XML Schema. This file defines the structure of all types and *Messages*. This schema captures normative names for types and their fields as well the order the fields appear when encoded. The data type of each field is also captured.

The normative contract for *Message* sent via the SOAP/HTTP *TransportProtocol* is a WSDL that includes XML Schema for the OPC UA XML encoded *Messages*. It also defines the port types for OPC UA *Servers* and *DiscoveryServers*.

Links to the WSDL and XML Schema files can be found in Annex D.

## Annex A (normative)

### Constants

#### A.1 Attribute Ids

Table A.1 – Identifiers assigned to Attributes

Attribute	Identifier
NodeId	1
NodeClass	2
BrowseName	3
DisplayName	4
Description	5
WriteMask	6
UserWriteMask	7
IsAbstract	8
Symmetric	9
InverseName	10
ContainsNoLoops	11
EventNotifier	12
Value	13
DataType	14
ValueRank	15
ArrayDimensions	16
AccessLevel	17
UserAccessLevel	18
MinimumSamplingInterval	19
Historizing	20
Executable	21
UserExecutable	22

#### A.2 Status Codes

This annex defines the numeric identifiers for all of the StatusCodes defined by the OPC UA Specification. The identifiers are specified in a CSV file with the following syntax:

```
<SymbolName>, <Code>, <Description>
```

Where the *SymbolName* is the literal name for the error code that appears in the specification and the *Code* is the hexadecimal value for the *StatusCode* (see IEC 62541-4). The severity associated with a particular code is specified by the prefix (*Good*, *Uncertain* or *Bad*).

The CSV released with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/schemas/1.02/StatusCode.csv>

NOTE The latest CSV that is compatible with this version of the standard can be found here:

<http://www.opcfoundation.org/UA/schemas/StatusCode.csv>

#### A.3 Numeric Node Ids

This annex defines the numeric identifiers for all of the numeric *NodeIds* defined by the OPC UA Specification. The identifiers are specified in a CSV file with the following syntax:

```
<SymbolName>, <Identifier>, <NodeClass>
```

Where the *SymbolName* is either the *BrowseName* of a *Type Node* or the *BrowsePath* for an *Instance Node* that appears in the specification and the *Identifier* is numeric value for the *NodeId*.

The *BrowsePath* for an instance *Node* is constructed by appending the *BrowseName* of the instance *Node* to *BrowseName* for the containing instance or type. A '\_' character is used to separate each *BrowseName* in the path. For example, IEC 62541-5 defines the *ServerType ObjectType Node* which has the *NamespaceArray Property*. The *SymbolName* for the *NamespaceArray InstanceDeclaration* within the *ServerType* declaration is: *ServerType\_NamespaceArray*. IEC 62541-5 also defines a standard instance of the *ServerType ObjectType* with the *BrowseName* 'Server'. The *BrowseName* for the *NamespaceArray Property* of the standard *Server Object* is: *Server\_NamespaceArray*.

The *NamespaceUri* for all *NodeIds* defined here is <http://opcfoundation.org/UA/>

The CSV released with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/schemas/1.02/NodeIds.csv>

NOTE The latest CSV that is compatible with this version of the standard can be found here:

<http://www.opcfoundation.org/UA/schemas/NodeIds.csv>

## **Annex B** (normative)

### **OPC UA Nodeset**

The OPC UA NodeSet includes the complete Information Model defined in this standard. It follows the XML Information Model schema syntax defined in Annex F and can thus be read and processed by a computer program.

The Information Model Schema released with this version of the standard can be found here:

<http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.NodeSet2.xml>

NOTE The latest Information Model schema that is compatible with this version of the standard can be found here:

<http://www.opcfoundation.org/UA/schemas/Opc.Ua.NodeSet2.xml>

## **Annex C** (normative)

### **Type declarations for the OPC UA native Mapping**

This Annex defines the OPC UA Binary encoding for all *DataTypes* and *Messages* defined in this standard. The schema used to describe the type is defined in IEC 62541-3.

The OPC UA Binary Schema released with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.Types.bsd.xml>

NOTE The latest file that is compatible with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/schemas/Opc.Ua.Types.bsd.xml>

## Annex D (normative)

### WSDL for the XML Mapping

#### D.1 XML Schema

This annex defines the XML Schema for all DataTypes and *Messages* defined in this series of OPC UA standards.

The XML Schema released with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.Types.xsd>

NOTE The latest file that is compatible with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/2008/02/Types.xsd>

#### D.2 WDSL Port Types

This annex defines the WSDL Operations and Port Types for all Services defined in IEC 62541-4.

The WSDL released with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.Services.wsdl>

NOTE The latest file that is compatible with this version of the standards can be found here:

<http://opcfoundation.org/UA/2008/02/Services.wsdl>

This WSDL imports the XML Schema defined in D.1.

#### D.3 WSDL Bindings

This annex defines the WSDL Bindings for all Services defined in IEC 62541-4.

The WSDL released with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/schemas/1.02/Opc.Ua.Endpoints.wsdl>

NOTE The latest file that is compatible with this version of the standards can be found here:

<http://opcfoundation.org/UA/2008/02/Endpoints.wsdl>

This WSDL imports the WSDL defined in D.2.

## Annex E (normative)

### Security settings management

#### E.1 Overview

All OPC UA applications shall support security; however, this requirement means that Administrators need to configure the security settings for the OPC UA *Application*. This appendix describes an XML Schema which can be used to read and update the security settings for a OPC UA *Application*. All OPC UA applications may support configuration by importing/exporting documents that conform to the schema (called the *SecuredApplication* schema) defined in this Annex.

The XML Schema released with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/schemas/1.02/SecuredApplication.xsd>

NOTE The latest file that is compatible with this version of this specification can be found here:

<http://opcfoundation.org/UA/2011/03/SecuredApplication.xsd>

The *SecuredApplication* schema can be supported in two ways:

- 1) Providing an XML configuration file that can be edited directly;
- 2) Providing a import/export utility that can be run as required;

If the *Application* supports direct editing of an XML configuration file then that file shall have exactly one element with the local name 'SecuredApplication' and URI equal to the *SecuredApplication* schema URI. A third party configuration utility shall be able to parse the XML file, read and update the 'SecuredApplication' element. The administrator shall ensure that only authorized administrators can update this file. The following is an example of a configuration that can be directly edited:

```
<s1:SampleConfiguration xmlns:s1="http://acme.com/UA/Sample/Configuration.xsd">
  <ApplicationName>ACME UA Server</ApplicationName>
  <ApplicationUri>urn:myfactory.com:Machine54:ACME UA Server</ApplicationUri>

  <!-- any number of application specific elements -->

  <SecuredApplication xmlns="http://opcfoundation.org/UA/2011/03/SecuredApplication.xsd">
    <ApplicationName>ACME UA Server</ApplicationName>
    <ApplicationUri>urn:myfactory.com:Machine54:ACME UA Server</ApplicationUri>
    <ApplicationType>Server_0</ApplicationType>
    <ApplicationCertificate>
      <StoreType>Windows</StoreType>
      <StorePath>LocalMachine\My</StorePath>
      <SubjectName>ACME UA Server</SubjectName>
    </ApplicationCertificate>
  </SecuredApplication>

  <!-- any number of application specific elements -->

  <DisableHiResClock>true</DisableHiResClock>
</s1:SampleConfiguration>
```

If an *Application* provides an import/export utility then the import/export file shall be a document that conforms to the *SecuredApplication* schema. The administrator shall ensure that only authorized administrators can run the utility. The following is an example of a file used by an import/export utility:

```
<?xml version="1.0" encoding="utf-8" ?>
<SecuredApplication xmlns="http://opcfoundation.org/UA/2011/03/SecuredApplication.xsd">
  <ApplicationName>ACME UA Server</ApplicationName>
```



```

<ApplicationUri>urn:myfactory.com:Machine54:ACME UA Server</ApplicationUri>
<ApplicationType>Server_0</ApplicationType>
<ConfigurationMode>urn:acme.com:ACME Configuration Tool</ConfigurationMode>
<LastExportTime>2011-03-04T13:34:12Z</LastExportTime>
<ExecutableFile>%ProgramFiles%\ACME\Bin\ACME UA Server.exe</ExecutableFile>
<ApplicationCertificate>
  <StoreType>Windows</StoreType>
  <StorePath>LocalMachine\My</StorePath>
  <SubjectName>ACME UA Server</SubjectName>
</ApplicationCertificate>
<TrustedCertificateStore>
  <StoreType>Windows</StoreType>
  <StorePath>LocalMachine\UA Applications</StorePath>
  <!-- Offline CRL Checks by Default -->
  <ValidationOptions>16</ValidationOptions>
</TrustedCertificateStore>
<TrustedCertificates>
  <Certificates>
    <CertificateIdentifier>
      <SubjectName>CN=MyFactory CA</SubjectName>
      <!-- Online CRL Check for this CA -->
      <ValidationOptions>32</ValidationOptions>
    </CertificateIdentifier>
  </Certificates>
</TrustedCertificates>
<RejectedCertificatesStore>
  <StoreType>Directory</StoreType>
  <StorePath>%CommonApplicationData%\OPC Foundation\RejectedCertificates</StorePath>
</RejectedCertificatesStore>
</SecuredApplication>

```

## E.2 SecuredApplication

The *SecuredApplication* element specifies the security settings for an *Application*. The elements contained in a *SecuredApplication* are described in Table E.1.

When an instance of a *SecuredApplication* is imported into an *Application* the *Application* updates its configuration based on the information contained within it. If unrecoverable errors occur during import an *Application* shall not make any changes to its configuration and report the reason for the error.

The mechanism used to import or export the configuration depends on the *Application*. Applications shall ensure that only authorized users are able to access this feature.

The *SecuredApplication* element may reference X509 Certificates which are contained in physical stores. Each *Application* needs to decide whether it uses shared physical stores which the administrator can control directly by changing the location or private stores that can only be accessed via the import/export utility. If the *Application* uses private stores then the contents of these private stores shall be copied to the export file during export. If the import file references shared physical stores then the import/export utility shall copy the contents of those stores to the private stores.

The import/export utility shall not export private keys. If the administrator wishes to assign a new public-private key to the *Application* the administrator shall place the private in a store where it can be accessed by the import/export utility. The import/export utility is then responsible for ensuring it is securely moved to a location where the *Application* can access it.

**Table E.1 – SecuredApplication**

Element	Type	Description
ApplicationName	String	A human readable name for the <i>Application</i> . Applications shall allow this value to be read or changed.
ApplicationUri	String	A globally unique identifier for the instance of the <i>Application</i> . Applications shall allow this value to be read or changed.
ApplicationType	ApplicationType	The type of <i>Application</i> . May be one of <ul style="list-style-type: none"> <li>• Server_0;</li> <li>• Client_1;</li> <li>• ClientAndServer_2;</li> <li>• DiscoveryServer_3;</li> </ul> <i>Application</i> shall provide this value. Applications do not allow this value to be changed.
ProductName	String	A name for the product. <i>Application</i> shall provide this value. Applications do not allow this value to be changed.
ConfigurationMode	String	Indicates how the <i>Application</i> should be configured. An empty or missing value indicates that the configuration file can be edited directly. The location of the configuration file shall be provided in this case. Any other value is a URI that identifies the configuration utility. The vendor documentation shall explain how to use this utility. <i>Application</i> shall provide this value. Applications do not allow this value to be changed.
LastExportTime	UtcTime	When the configuration was exported by the import/export utility. It may be omitted if Applications allow direct editing of the security configuration.
ConfigurationFile	String	The full path to a configuration file used by the <i>Application</i> . Applications do not provide this value if a import/export utility is used. Applications do not allow this value to be changed. Permissions set on this file shall control who has rights to change the configuration of the <i>Application</i> .
ExecutableFile	String	The full path to an executable file for the <i>Application</i> . Applications may not provide this value. Applications do not allow this value to be changed. Permissions set on this file shall control who has rights to launch the <i>Application</i> .
ApplicationCertificate	CertificateIdentifier	The identifier for the <i>ApplicationInstance Certificate</i> . Applications shall allow this value to be read or changed. This identifier may reference a <i>Certificate</i> store that contains the private key. If the private key is not accessible to outside applications this value shall contain the X509 <i>Certificate</i> for the <i>Application</i> . If the configuration utility assigns a new private key this value shall reference the store where the private key is placed. The import/export utility may delete this private key if it moves it to a secure location accessible to the <i>Application</i> . Applications shall allow Administrators to enter the password required to access the private key during the import operation. The exact mechanism depends on the <i>Application</i> . Applications shall report an error if the ApplicationCertificate is not valid.

Element	Type	Description
TrustedCertificateStore	CertificateStore Identifier	<p>The location of the CertificateStore containing the Certificates of Applications or <i>Certificate</i> Authorities (CAs) which can be trusted. Applications shall allow this value to be read or changed.</p> <p>This value shall be a reference to a physical store which can be managed separately from the <i>Application</i>. Applications that support shared physical stores shall check this store for changes whenever they validate a <i>Certificate</i>.</p> <p>The Administrator is responsible for verifying the signature on all Certificates placed in this store. This means the <i>Application</i> may trust Certificates in this store even if they cannot be verified back to a trusted root.</p> <p>Administrators shall place any CA certificates used to verify the signature in the UntrustedIssuerStore or the UntrustedIssuerList. This will allow applications to properly verify the signatures.</p> <p>The <i>Application</i> shall check the revocation status of the Certificates in this store if the <i>Certificate</i> was issued by a CA. The <i>Application</i> shall look for the offline <i>Certificate</i> Revocation List (CRL) for a CA in the store where it found the CA <i>Certificate</i>.</p> <p>The location of an online CRL for CA shall be specified with the CRLDistributionPoints (OID= 2.5.29.31) X509 <i>Certificate</i> extension.</p> <p>The ValidationOptions parameter is used to specify which revocation list should be used for CAs in this store.</p>
TrustedCertificates	CertificateList	<p>A list of Certificates for Applications for CAs that can be trusted. Applications shall allow this value to be read or changed.</p> <p>The value is an explicit list of Certificates which is private to the <i>Application</i>. It is used when the <i>Application</i> does not support shared physical <i>Certificate</i> stores or when Administrators need to specify ValidationOptions for individual Certificates.</p> <p>If the TrustedCertificateStore and the TrustedCertificates parameters are both specified then the <i>Application</i> shall use the TrustedCertificateStore for checking trust relationships. The TrustedCertificates parameter is only used to lookup ValidationOptions for individual Certificates. It may also be used to provide CRLs for CA certificates.</p> <p>If the TrustedCertificateStore is not specified then TrustedCertificates parameter shall contain the complete X509 <i>Certificate</i> for each entry.</p>
IssuerStore	CertificateStore Identifier	<p>The location of the CertificateStore containing CA Certificates which are not trusted but are needed to check signatures on Certificates. Applications shall allow this value to be read or changed.</p> <p>This value shall be a reference to a physical store which can be managed separately from the <i>Application</i>. Applications that support shared physical stores shall check this store for changes whenever they validate a <i>Certificate</i>.</p> <p>This store may also contain CRLs for the CAs.</p>
IssuerCertificates	CertificateList	<p>A list of Certificates for CAs which are not trusted but are needed to check signatures on Certificates. Applications shall allow this value to be read or changed.</p> <p>The value is an explicit list of Certificates which is private to the <i>Application</i>. It is used when the <i>Application</i> does not support shared physical <i>Certificate</i> stores or when Administrators need to specify ValidationOptions for individual Certificates.</p> <p>If the IssuerStore and the IssuerCertificates parameters are both specified then the <i>Application</i> shall use the IssuerStore for checking signatures. The IssuerCertificates parameter is only used to lookup ValidationOptions for individual Certificates. It may also be used to provide CRLs for CA certificates.</p>
RejectedCertificatesStore	CertificateStore Identifier	<p>The location of the shared CertificateStore containing the Certificates of Applications which were rejected. Applications shall allow this value to be read or changed.</p> <p>Applications shall add the DER encoded <i>Certificate</i> into this store whenever it rejects a <i>Certificate</i> because it is untrusted or if it failed one of the validation rules which can be suppressed (see Clause E.6).</p> <p>Applications shall not add a <i>Certificate</i> to this store if it was rejected for a reason that cannot be suppressed (e.g. <i>Certificate</i> revoked).</p>

Element	Type	Description
BaseAddresses	String[]	A list of URLs for the <i>Endpoints</i> supported by a <i>Server</i> . Applications shall allow these values to be read or changed. If a <i>Server</i> does not support the scheme for a URL it shall ignore it. This list can have multiple entries for the same URL scheme. The first entry for a scheme is the base URL. The rest are assumed to be DNS aliases that point to the first URL. It is the responsibility of the Administrator to configure the network to route these aliases correctly.
SecurityProfileUris	SecurityProfile[] ProfileUri String Enabled Boolean	A list of security profiles supported by a <i>Server</i> . Applications shall allow these values to be read or changed. Applications shall allow the Enabled flag to be changed for each <i>SecurityProfile</i> that it supports. If the Enabled flag is false the <i>Server</i> shall not allow connections using the <i>SecurityProfile</i> . If a <i>Server</i> does not support a <i>SecurityProfile</i> it shall ignore it.
Extensions	xs:any	A list of vendor defined Extensions attached to the security settings. Applications shall ignore Extensions that they do not recognize. Applications that update a file containing Extensions shall not delete or modify extensions that they do not recognize.

### E.3 CertificateIdentifier

The *CertificateIdentifier* element describes an X509 *Certificate*. The *Certificate* can be provided explicitly within the element or the element can specify the location of the *CertificateStore* that contains the *Certificate*. The elements contained in a *CertificateIdentifier* are described in Table E.2.

**Table E.2 – CertificateIdentifier**

Element	Type	Description
StoreType	String	The type of <i>CertificateStore</i> that contains the <i>Certificate</i> . Predefined values are "Windows" and "Directory". If not specified the <i>RawData</i> element shall be specified.
StorePath	String	The path to the <i>CertificateStore</i> . The syntax depends on the <i>StoreType</i> . If not specified the <i>RawData</i> element shall be specified.
SubjectName	String	The <i>SubjectName</i> for the <i>Certificate</i> . The Common Name (CN) component of the <i>SubjectName</i> . The <i>SubjectName</i> represented as a string that complies with Section 3 of RFC 4514. Values that do not contain '=' characters are presumed to be the Common Name component.
Thumbprint	String	The SHA1 thumbprint for the <i>Certificate</i> formatted as a hexadecimal string. Case is not significant.
RawData	ByteString	The DER encoded <i>Certificate</i> . The <i>CertificateIdentifier</i> is invalid if the information in the DER <i>Certificate</i> conflicts with the information specified in other fields. Import utilities shall reject configurations containing invalid <i>Certificates</i> . This field shall not be specified if the <i>StoreType</i> and <i>StorePath</i> are specified.
ValidationOptions	Int32	The options to use when validating the <i>Certificate</i> . The possible options are described in E.6.
OfflineRevocationList	ByteString	A <i>Certificate</i> Revocation List (CRL) associated with an Issuer <i>Certificate</i> . The format of a CRL is defined by RFC 3280. This field is only meaningful for Issuer <i>Certificates</i> .
OnlineRevocationList	String	A URL for an Online Revocation List associated with an Issuer <i>Certificate</i> . This field is only meaningful for Issuer <i>Certificates</i> .

A "Windows" *StoreType* specifies a Windows *Certificate* store.

The syntax of the StorePath has the form:

[\\HostName\]StoreLocation[(\ServiceName | UserSid)]\StoreName

where:

HostName – the name of the machine where the store resides.

StoreLocation – one of LocalMachine, CurrentUser, User or Service

ServiceName – the name of a Windows Service.

UserSid – the SID for a Windows user account.

StoreName – the name of the store (e.g. My, Root, Trust, CA, etc.).

Examples of Windows StorePaths are:

\\MYPC\LocalMachine\My

\CurrentUser\Trust

\\MYPC\Service\My UA Server\UA Applications

\User\S-1-5-25\Root

A "Directory" StoreType specifies a directory on disk which contains files with DER encoded Certificates. The name of the file is the SHA1 thumbprint for the *Certificate*. Only public keys may be placed in a "Directory" Store. The StorePath is an absolute file system path with a syntax that depends on the operating system.

If a "Directory" store contains a 'certs' subdirectory then it is presumed to be a structured store with the subdirectories described in Table E.3.

**Table E.3 – Structured directory store**

Subdirectory	Description
certs	Contains the DER encoded X509 Certificates. The files shall have a .der file extension.
private	Contains the private keys. The format of the file may be <i>Application</i> specific. PEM encoded files should have a .pem extension. PKCS#12 encoded files should have a .pfx extension. The root file name shall be the same as the corresponding public key file in the certs directory.
crl	Contains the DER encoded CRL for any CA Certificates found in the certs or ca directories. The files shall have a .crl file extension.

Each *Certificate* is uniquely identified by its Thumbprint. The SubjectName or the distinguished SubjectName may be used to identify a *Certificate* to a human; however, they are not unique. The SubjectName may be specified in conjunction with the Thumbprint or the RawData. If there is an inconsistency between the information provided then the *CertificateIdentifier* is invalid. Invalid *CertificateIdentifiers* are handled differently depending on where they are used.

It is recommended that the SubjectName always be specified.

A *Certificate* revocation list (CRL) contains a list of certificates issued by a CA that are no longer trusted. These lists should be checked before an *Application* can trust a *Certificate* issued by a trusted CA. The format of a CRL is defined by RFC 3280.

Offline CRLs are placed in a local *Certificate* store with the Issuer *Certificate*. Online CRLs may exist but the protocol depends on the system. An online CRL is identified by a URL.

#### E.4 CertificateStoreIdentifier

The *CertificateStoreIdentifier* element describes a physical store containing X509 Certificates. The elements contained in a *CertificateStoreIdentifier* are described in Table E.4.

**Table E.4 – CertificateStoreIdentifier**

Element	Type	Description
StoreType	String	The type of CertificateStore that contains the <i>Certificate</i> . Predefined values are "Windows" and "Directory".
StorePath	String	The path to the CertificateStore. The syntax depends on the StoreType. See E.3 for a description of the syntax for different StoreTypes.
ValidationOptions	Int32	The options to use when validating the Certificates contained in the store. The possible options are described in E.6.

All *Certificates* are placed in a physical store which can be protected from unauthorized access. The implementation of a store can vary and will depend on the *Application*, development tool or operating system. A *Certificate* store may be shared by many applications on the same machine.

Each *Certificate* store is identified by a *StoreType* and a *StorePath*. The same path on different machines identifies a different store.

#### E.5 CertificateList

The *CertificateList* element is a list of *Certificates*. The elements contained in a *CertificateList* are described in Table E.5.

**Table E.5 – CertificateList**

Element	Type	Description
Certificates	CertificateIdentifier[]	The list of Certificates contained in the Trust List
ValidationOptions	Int32	The options to use when validating the Certificates contained in the store. These options only apply to <i>Certificates</i> that have <i>ValidationOptions</i> with the <i>UseDefaultOptions</i> bit set. The possible options are described in E.6.

#### E.6 CertificateValidationOptions

The *CertificateValidationOptions* control the process used to validate a *Certificate*. Any *Certificate* can have validation options associated. If none are specified the *ValidationOptions* for the store or list containing the *Certificate* are used. The possible options are shown in Table E.6.

**Table E.6 – CertificateValidationOptions**

Field	Bit	Description
SuppressCertificateExpired	0	Ignore errors related to the validity time of the <i>Certificate</i> or its issuers.
SuppressHostNameInvalid	1	Ignore mismatches between the host name or <i>Application uri</i> .
SuppressRevocationStatusUnknown	2	Ignore errors if the issuer's revocation list cannot be found.
CheckRevocationStatusOnline	3	<p>Check the revocation status online.</p> <p>If set the validator will look for the URL of the CRL Distribution Point in the <i>Certificate</i> and use the OCSP (Online Certificate Status Protocol) to determine if the <i>Certificate</i> has been revoked.</p> <p>If the CRL Distribution Point is not reachable then the validator will look for offline CRLs if the <i>CheckRevocationStatusOffline</i> bit is set. Otherwise, validation fails.</p> <p>This option is specified for Issuer <i>Certificates</i> and used when validating <i>Certificates</i> issued by that Issuer.</p>
CheckRevocationStatusOffline	4	<p>Check the revocation status offline.</p> <p>If set the validator will look a CRL in the <i>Certificate Store</i> where the CA <i>Certificate</i> was found.</p> <p>Validation fails if a CRL is not found.</p> <p>This option is specified for Issuer <i>Certificates</i> and used when validating <i>Certificates</i> issued by that <i>Issuer</i>.</p>
UseDefaultOptions	5	<p>If set the <i>CertificateValidationOptions</i> from the <i>CertificateList</i> shall be used.</p> <p>If a <i>Certificate</i> does not belong to a <i>CertificateList</i> then the default is 0 for all bits.</p>

## Annex F (normative)

### Information Model XML Schema

#### F.1 Overview

Information Model developers define standard *AddressSpaces* which are implemented by many *Servers*. There is a need for a standard syntax that Information Model developers can use to formally define their models in a form that can be read by a computer program. This Annex defines an XML-based schema for this purpose.

The XML Schema released with this version of the standards can be found here:

<http://www.opcfoundation.org/UA/schemas/1.02/UANodeSet.xsd>

NOTE The latest file that is compatible with this version of the standards can be found here:

<http://opcfoundation.org/UA/2011/03/UANodeSet.xsd>

The schema document is the formal definition. The description in this Annex only discusses details of the semantics that cannot be captured in the schema document. Types which are self-describing are not discussed.

This schema can also be used to serialize (i.e. import or export) an arbitrary set of *Nodes* in the *Server Address Space*. This serialized form can be used to save *Server* state for use by the *Server* later or to exchange with other applications (e.g. to support offline configuration by a *Client*).

#### F.2 UANodeSet

The *UANodeSet* is the root of the document. It defines a set of *Nodes*, their *Attributes* and *References*. *References* to *Nodes* outside of the document are allowed.

The structure of a *UANodeSet* is shown in Table F.1.

**Table F.1 – UANodeSet**

Element	Type	Description
NamespaceUri	UriTable	A list of <i>NamespaceUri</i> used in the <i>UANodeSet</i> .
ServerUri	UriTable	A list of <i>ServerUri</i> used in the <i>UANodeSet</i> .
Aliases	AliasTable	A list of <i>Aliases</i> used in the <i>UANodeSet</i> .
Extensions	xs:any	An element containing any vendor defined extensions to the <i>UANodeSet</i> .
<choice>	UObject UVariable UAMethod UAView UObjectType UVariableType UADatatype UReferenceType	The <i>Nodes</i> in the <i>UANodeSet</i> .

The *NamespaceUri* is a list of URIs for namespaces used in the *UANodeSet*. The *NamespaceIndexes* used in *NodeId*, *ExpandedNodeIds* and *QualifiedNames* identify an element in this list. The first index is always 1 (0 is always the OPC UA namespace).



The *ServerUris* is a list of URIs for *Servers* referenced in the *UANodeSet*. The *ServerIndex* in *ExpandedNodeIds* identifies an element in this list. The first index is always 1 (0 is always the current *Server*).

The *Aliases* are a list of string substitutions for *NodeIds*. *Aliases* can be used to make the file more readable by allowing a string like 'HasProperty' in place of a numeric *NodeId* (i=46). *Aliases* are optional.

The *Extensions* are free form XML data that can be used to attach vendor defined data to the *UANodeSet*.

### F.3 UANode

A *UANode* is an abstract base type for all *Nodes*. It defines the base set of *Attributes* and the *References*. There are subtypes for each *NodeClass* defined in IEC 62541-4. Each of these subtypes defines XML elements and attributes for the OPC UA *Attributes* specific to the *NodeClass*. The fields in the *UANode* type are defined in Table F.2.

**Table F.2 – UANode**

Element	Type	Description
NodeId	NodeId	A <i>NodeId</i> serialized as a <i>String</i> . The syntax of the serialized <i>String</i> is defined in 5.3.1.10.
BrowseName	QualifiedName	A <i>QualifiedName</i> serialized as a <i>String</i> with the form: <namespace index>:<name> Where the <i>NamespaceIndex</i> refers to the <i>NamespaceUris</i> table.
SymbolicName	String	A symbolic name for the <i>Node</i> that can be used as a class/field name in autogenerated code. It should only be specified if the <i>BrowseName</i> cannot be used for this purpose. This field does not appear in the <i>AddressSpace</i> and is intended for use by design tools. Only letters, digits or the underscore ('_') are permitted.
WriteMask	WriteMask	The value of the <i>WriteMask</i> Attribute.
UserWriteMask	WriteMask	The value of the <i>UserWriteMask</i> Attribute.
DisplayName	LocalizedText[]	A list of <i>DisplayNames</i> for the <i>Node</i> in different locales. There shall be only one entry per locale.
Description	LocalizedText[]	The list of the <i>Descriptions</i> for the <i>Node</i> in different locales. There shall be only one entry per locale.
References	Reference[]	The list of <i>References</i> for the <i>Node</i> .
Extensions	xs:any	An element containing any vendor defined extensions to the <i>UANode</i> .

The *Extensions* are free form XML data that can be used to attach vendor defined data to the *UANode*.

### F.4 Reference

The *Reference* type specifies a *Reference* for a *Node*. The *Reference* can be forward or inverse. Only one direction for each *Reference* needs to be in a *UANodeSet*. The other direction shall be added automatically during any import operation. The fields in the *Reference* type are defined in Table F.3.

**Table F.3 – Reference**

Element	Type	Description
NodeId	NodeId	The <i>NodeId</i> of the target of the <i>Reference</i> serialized as a <i>String</i> . The syntax of the serialized <i>String</i> is defined in 5.3.1.11 ( <i>ExpandedNodeId</i> ). This value can be replaced by an <i>Alias</i> .
ReferenceType	NodeId	The <i>NodeId</i> of the <i>ReferenceType</i> serialized as a <i>String</i> . The syntax of the serialized <i>String</i> is defined in 5.3.1.10 ( <i>NodeId</i> ). This value can be replaced by an <i>Alias</i> .
IsForward	Boolean	If TRUE the <i>Reference</i> is a forward reference.

## F.5 UAType

A *UAType* is a subtype of the *UANode* defined in F.3. It is the base type for the types defined in Table F.4.

**Table F.4 – UANodeSet Type Nodes**

Subtype	Description
UAObjectType	Defines an <i>ObjectType Node</i> as described in IEC 62541-3.
UAVariableType	Defines a <i>VariableType Node</i> as described in IEC 62541-3.
UADataType	Defines a <i>Data Type Node</i> as described in IEC 62541-3.
UReferenceType	Defines a <i>ReferenceType Node</i> as described in IEC 62541-3.

## F.6 UAInstance

A *UAInstance* is a subtype of the *UANode* defined in F.3. It is the base type for the types defined in Table F.5. The fields in the *UAInstance* type are defined in Table F.6.

**Table F.5 – UANodeSet Instance Nodes**

Subtype	Description
UAObject	Defines an <i>Object Node</i> as described in IEC 62541-3.
UAVariable	Defines a <i>Variable Node</i> as described in IEC 62541-3.
UAMethod	Defines a <i>Method Node</i> as described in IEC 62541-3.
UAView	Defines a <i>View Node</i> as described in IEC 62541-3.

**Table F.6 – UAInstance**

Element	Type	Description
All of the fields from the <i>UANode</i> type described in F.3.		
ParentNodeId	NodeId	The <i>NodeId</i> of the <i>Node</i> that is the parent of the <i>Node</i> within the information model. This field is used to indicate that a tight coupling exists between the <i>Node</i> and its parent (e.g. when the parent is deleted the child is deleted as well). This information does not appear in the <i>AddressSpace</i> and is intended for use by design tools.

## F.7 UAVariable

A *UAVariable* is a subtype of the *UAInstance* defined in F.6. It represents a Variable Node. The fields in the *UAVariable* type are defined in Table F.7.

Table F.7 – UVariable

Element	Type	Description
All of the fields from the <i>UInstance</i> type described in 0.		
Value	Variant	The Value of the Node encoding using the UA XML wire encoding.
Translation	TranslationType[]	A list of translations for the Value if the Value is a LocalizedText or a structure containing LocalizedTexts. This field may be omitted. If the Value is an array the number of elements in this array shall match the number of elements in the Value. Extra elements are ignored. If the Value is a scalar then there is one element in this array. If the Value is a structure then the each element contains translations for one or more fields identified by a name. See the TranslationType for more information.
DataType	NodeId	The data type of the value.
ValueRank	ValueRank	The value rank.
ArrayDimensions	ArrayDimensions	The number of dimensions in an array value.
AccessLevel	AccessLevel	The access level.
UserAccessLevel	AccessLevel	The access level for the current user.
MinimumSamplingInterval	Duration	The minimum sampling interval.
Historizing	Boolean	Whether history is being archived.

## F.8 UAMethod

A *UAMethod* is a subtype of the *UInstance* defined in 0. It represents a Method Node. The fields in the *UAMethod* type are defined in Table F.8.

Table F.8 – UAMethod

Element	Type	Description
All of the fields from the <i>UInstance</i> type described in 0.		
MethodDeclarationId	NodeId	May be specified for <i>Method Nodes</i> that are a target of a <i>HasComponent</i> reference from a single <i>Object Node</i> . It is the <i>NodeId</i> of the <i>UAMethod</i> with the same <i>BrowseName</i> contained in the <i>TypeDefinition</i> associated with the <i>Object Node</i> . If the <i>TypeDefinition</i> overrides a <i>Method</i> inherited from a base <i>ObjectType</i> then this attribute shall reference the <i>Method Node</i> in the subtype.

## F.9 TranslationType

A *TranslationType* contains additional translations for *LocalizedTexts* used in the *Value* of a *Variable*. The fields in the *TranslationType* are defined in Table F.9. If multiple *Arguments* existed there would be a Translation element for each *Argument*.

The type can have two forms depending on whether the *Value* is a *LocalizedText* or a *Structure* containing *LocalizedTexts*. If it is a *LocalizedText* it contains a simple list of translations. If it is a *Structure* it contains a list of fields which each contain a list of translations. Each field is identified by a Name which is unique within the structure. The mapping between the Name and the *Structure* requires an understanding of the *Structure* encoding. If the *Structure* field is encoded as a *LocalizedText* with UA XML then the name is the unqualified path to the XML element where names in the path are separated by '/'. For example, a structure with a nested structure containing a *LocalizedText* could have a path like "Server/ApplicationName".

The following example illustrates how translations for the Description field in the *Argument Structure* are represented in XML:

```
<Value>
  <ListOfExtensionObject xmlns="http://opcfoundation.org/UA/2008/02/Types.xsd">
```

```

<ExtensionObject>
  <TypeId>
    <Identifier>i=297</Identifier>
  </TypeId>
  <Body>
    <Argument>
      <Name>ConfigData</Name>
      <DataType>
        <Identifier>i=15</Identifier>
      </DataType>
      <ValueRank>-1</ValueRank>
      <ArrayDimensions />
      <Description>
        <Text>[English Translation for Description]</Text>
      </Description>
    </Argument>
  </Body>
</ExtensionObject>
</ListOfExtensionObject>
</Value>
<Translation>
  <Field Name="Description">
    <Text Locale="de-DE">[German Translation for Description]</Text>
    <Text Locale="fr-FR">[French Translation for Description]</Text>
  </Field>
</Translation>

```

If multiple Arguments existed there would be a Translation element for each Argument.

**Table F.9 – TranslationType**

Element	Type	Description
Text	LocalizedText[]	An array of translations for the Value. It only appears if the Value is a LocalizedText or an array of LocalizedText.
Field	StructureTranslationType[]	An array of structure fields which have translations. It only appears if the Value is a Structure or an array of Structures.
Name	String	The name of the field. This uniquely identifies the field within the structure. The exact mapping depends on the encoding of the structure.
Text	LocalizedText[]	An array of translations for the structure field.

## F.10 UADatatype

A *UADatatype* is a subtype of the *UAType* defined in 0. It defines a *Data Type Node*. The fields in the *UADatatype* type are defined in Table F.10.

**Table F.10 – UADatatype**

Element	Type	Description
All of the fields from the <i>UANode</i> type described in F.3.		
Definition	DataTypeDefinition	An abstract definition of the data type that can be used by design tools to create code that can serialize the data type in XML and/or Binary forms. It does not appear in the <i>AddressSpace</i> . This is only used to define subtypes of the <i>Structure</i> or <i>Enumeration Data Types</i> .

## F.11 DataTypeDefinition

A *DataTypeDefinition* defines an abstract representation of a *UADatatype* that can be used by design tools to automatically create serialization code. The fields in the *DataTypeDefinition* type are defined in Table F.11.

**Table F.11 – DataTypeDefinition**

Element	Type	Description
Name	QualifiedName	A unique name for the data type. This field is only specified for nested <i>DataTypeDefinitions</i> . The <i>BrowseName</i> of the <i>DataType Node</i> is used otherwise.
SymbolicName	String	A symbolic name for the data type that can be used as a class/structure name in autogenerated code. It should only be specified if the <i>Name</i> cannot be used for this purpose. Only letters, digits or the underscore ('_') are permitted. This field is only specified for nested <i>DataTypeDefinitions</i> . The <i>SymbolicName</i> of the <i>DataType Node</i> is used otherwise.
BaseType	QualifiedName	The name of any base type. Note that the <i>BaseType</i> can refer to types defined in other files. The <i>NamespaceUri</i> associated with the <i>Name</i> should indicate where to look for the <i>BaseType</i> definition. This field is only specified for nested <i>DataTypeDefinitions</i> . The <i>HasSubtype Reference</i> of the <i>DataType Node</i> is used otherwise.
Fields	DataTypeField[]	The list of fields that make up the data type. This definition assumes the structure has a sequential layout. For enumerations the fields are simply a list of values. Unions are not supported.

## F.12 DataTypeField

A *DataTypeField* defines an abstract representation of a field within a *UADatatype* that can be used by design tools to automatically create serialization code. The fields in the *DataTypeField* type are defined in Table F.12.

**Table F.12 – DataTypeField**

Element	Type	Description
Name	String	A name for the field that is unique within the <i>DataTypeDefinition</i> .
SymbolicName	String	A symbolic name for the field that can be used in autogenerated code. It should only be specified if the <i>Name</i> cannot be used for this purpose. Only letters, digits or the underscore ('_') are permitted.
DataType	NodeId	The <i>NodeId</i> of the <i>DataType</i> for the field. This <i>NodeId</i> can refer to another <i>Node</i> with its own <i>DataTypeDefinition</i> . This field is not specified for subtypes of <i>Enumeration</i> .
ValueRank	Int32	The value rank for the field. It shall be <i>Scalar</i> (-1) or a fixed rank <i>Array</i> (>=1). This field is not specified for subtypes of <i>Enumeration</i> .
Description	LocalizedText[]	A description for the field in multiple locales.
Definition	DataTypeDefinition	The field is a structure with a layout specified by the definition. This field is optional. This field allows designers to create nested structures without defining a new <i>DataType Node</i> for each structure. This field is not specified for subtypes of <i>Enumeration</i> .
Value	Int32	The value associated with the field. This field is only specified for subtypes of <i>Enumeration</i> .

## F.13 Variant

The *Variant* type specifies the value for a *Variable* or *VariableType Node*. This type is the same as the type defined in 5.3.1.17. As a result, the functions used to serialize *Variants* during *Service* calls can be used to serialize *Variant* in this file syntax.

*Variants* can contain *NodeIds*, *ExpandedNodeIds* and *QualifiedNames* which must be modified so the *NamespaceIndexes* and *ServerIndexes* reference the *NamespaceUri* and *ServerUri* tables in the *UANodeSet*.

*Variants* can also contain *ExtensionObjects* which contain an *EncodingId* and a *Structure* with fields which could be *NodeIds*, *ExpandedNodeIds* or *QualifiedNames*. The *NamespaceIndexes* and *ServerIndexes* in these fields shall also reference the tables in the *UANodeSet*.

## F.14 Example (Informative)

An example of the *UANodeSet* can be found below.

This example defines the *Nodes* for an *InformationModel* with the URI of “http://sample.com/Instances”. This example references *Nodes* defined in the base OPC UA *InformationModel* and an *InformationModel* with the URI “http://sample.com/Types”.

The XML namespaces declared at the top include the URIs for the *Namespaces* referenced in the document because the document includes *Complex Data*. Documents without *Complex Data* would not have these declarations.

```
<UANodeSet
xmlns:s1="http://sample.com/Instances"
xmlns:s0="http://sample.com/Types"
xmlns:uax="http://opcfoundation.org/UA/2008/02/Types.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://opcfoundation.org/UA/2011/03/UANodeSet.xsd">
```

The *NamespaceUris* table includes all *Namespaces* referenced in the document except for the base OPC UA *InformationModel*. A *NamespaceIndex* of 1 refers to the URI “http://sample.com/Instances”.

```
<NamespaceUris>
  <Uri>http://sample.com/Instances</Uri>
  <Uri>http://sample.com/Types</Uri>
</NamespaceUris>
```

The *Aliases* table is provided to enhance readability. There are no rules for what is included. A useful guideline would include standard *ReferenceTypes* and *DataTypes* if they are referenced in the document.

```
<Aliases>
  <Alias Alias="HasComponent">i=47</Alias>
  <Alias Alias="HasProperty">i=46</Alias>
  <Alias Alias="HasSubtype">i=45</Alias>
  <Alias Alias="HasTypeDefinition">i=40</Alias>
</Aliases>
```

The *BicycleType* is a *Data Type Node* that inherits from a *Data Type* defined in another *InformationModel* (ns=2;i=314). It is assumed that any *Application* importing this file will already know about the referenced *InformationModel*. A *Server* could map the references onto another OPC UA *Server* by adding a *ServerIndex* to *TargetNode NodeIds*. The structure of the *Data Type* is defined by the *Definition* element. This information can be used by code generators to automatically create serializers for the *Data Type*.

```
<UADataType NodeId="ns=1;i=365" BrowseName="1:BicycleType">
  <DisplayName>BicycleType</DisplayName>
  <References>
    <Reference ReferenceType="HasSubtype" IsForward="false">ns=2;i=314</Reference>
  </References>
```

```

<Definition Name="BicycleType" BaseType="0:1:BicycleType">
  <Field Name="NoOfGears" DataType="UInt32" />
  <Field Name="ManufacturerName" DataType="QualifiedName" />
</Definition>
</UADataType>

```

This *Node* is an instance of an *Object TypeDefinition Node* defined in another *InformationModel* (ns=2;i=341). It has a single *Property* which is declared later in the document.

```

<UAObject NodeId="ns=1;i=375" BrowseName="1:DriverOfTheMonth" ParentNodeId="ns=1;i=281">
  <DisplayName>DriverOfTheMonth</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty">ns=1;i=376</Reference>
    <Reference ReferenceType="HasTypeDefinition">ns=2;i=341</Reference>
    <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=281</Reference>
  </References>
</UAObject>

```

This *Node* is an instance of a *Variable TypeDefinition Node* defined in base OPC UA *InformationModel* (i=68). The *DataType* is the base type for the *BicycleType DataType*. The *AccessLevels* declare the *Variable* as *Readable* and *Writeable*. The *ParentNodeId* indicates that this *Node* is tightly coupled with the Parent (*DriverOfTheMonth*) and will be deleted if the Parent is deleted.

```

<UAVariable NodeId="ns=1;i=376" BrowseName="2:PrimaryVehicle"
  ParentNodeId="ns=1;i=375" DataType="ns=2;i=314" AccessLevel="3" UserAccessLevel="3">
  <DisplayName>PrimaryVehicle</DisplayName>
  <References>
    <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
    <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=375</Reference>
  </References>

```

This *Value* is an instance of a *BicycleType DataType*. It is wrapped in an *ExtensionObject* which declares that the value is serialized using the *Default XML DataTypeEncoding* for the *DataType*. The *Value* could be serialized using the *Default Binary DataTypeEncoding* but that would result in a document that cannot be edited by hand. No matter which *DataTypeEncoding* is used, the *NamespaceIndex* used in the *ManufacturerName* field refers to the *NamespaceUri* table in this document. The *Application* is responsible for changing whatever value it needs to be when the document is loaded by an *Application*.

```

<Value>
  <ExtensionObject xmlns="http://opcfoundation.org/UA/2008/02/Types.xsd">
    <TypeId>
      <Identifier>ns=1;i=366</Identifier>
    </TypeId>
    <Body>
      <s1:BicycleType>
        <s0:Make>Trek</s0:Make>
        <s0:Model>Compact</s0:Model>
        <s1:NoOfGears>10</s1:NoOfGears>
        <s1:ManufacturerName>
          <uax:NamespaceIndex>1</uax:NamespaceIndex>
          <uax:Name>Hello</uax:Name>
        </s1:ManufacturerName>
      </s1:BicycleType>
    </Body>
  </ExtensionObject>
</Value>
</UAVariable>

```

These are the *DataTypeEncoding Nodes* for the *BicycleType DataType*.

```

<UAObject NodeId="ns=1;i=366" BrowseName="Default XML">
  <DisplayName>Default XML</DisplayName>
  <References>
    <Reference ReferenceType="HasEncoding" IsForward="false">ns=1;i=365</Reference>
    <Reference ReferenceType="HasDescription">ns=1;i=367</Reference>
    <Reference ReferenceType="HasTypeDefinition">i=76</Reference>
  </References>
</UAObject>

```

```

<UAObject NodeId="ns=1;i=370" BrowseName="Default Binary">
  <DisplayName>Default Binary</DisplayName>
  <References>
    <Reference ReferenceType="HasEncoding" IsForward="false">ns=1;i=365</Reference>
    <Reference ReferenceType="HasDescription">ns=1;i=371</Reference>
    <Reference ReferenceType="HasTypeDefinition">i=76</Reference>
  </References>
</UAObject>

```

This is the *DataTypeDescription Node* for the *Default XML DataTypeEncoding* of the *BicycleType DataType*. The *Value* is one of the built-in types.

```

<UAVariable NodeId="ns=1;i=367" BrowseName="1:BicycleType" DataType="String">
  <DisplayName>BicycleType</DisplayName>
  <References>
    <Reference ReferenceType="HasTypeDefinition">i=69</Reference>
    <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=341</Reference>
  </References>
  <Value>
    <uax:String>//xs:element[@name='BicycleType']</uax:String>
  </Value>
</UAVariable>

```

This is the *DataTypeDescription Node* for the *DataTypeDescription* declared above. The XML Schema document is a UTF-8 document stored as a base64 string. This allows clients to read the schema for the

```

<UAVariable NodeId="ns=1;i=341" BrowseName="1:Quickstarts.DataTypes.Instances"
DataType="ByteString">
  <DisplayName>Quickstarts.DataTypes.Instances</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty">ns=1;i=343</Reference>
    <Reference ReferenceType="HasComponent">ns=1;i=367</Reference>
    <Reference ReferenceType="HasComponent" IsForward="false">i=92</Reference>
    <Reference ReferenceType="HasTypeDefinition">i=72</Reference>
  </References>
  <Value>
    <uax:ByteString>PHhz...WlhPg==</uax:ByteString>
  </Value>
</UAVariable>

```

---





# British Standards Institution (BSI)

BSI is the national body responsible for preparing British Standards and other standards-related publications, information and services.

BSI is incorporated by Royal Charter. British Standards and other standardization products are published by BSI Standards Limited.

## About us

We bring together business, industry, government, consumers, innovators and others to shape their combined experience and expertise into standards-based solutions.

The knowledge embodied in our standards has been carefully assembled in a dependable format and refined through our open consultation process. Organizations of all sizes and across all sectors choose standards to help them achieve their goals.

## Information on standards

We can provide you with the knowledge that your organization needs to succeed. Find out more about British Standards by visiting our website at [bsigroup.com/standards](http://bsigroup.com/standards) or contacting our Customer Services team or Knowledge Centre.

## Buying standards

You can buy and download PDF versions of BSI publications, including British and adopted European and international standards, through our website at [bsigroup.com/shop](http://bsigroup.com/shop), where hard copies can also be purchased.

If you need international and foreign standards from other Standards Development Organizations, hard copies can be ordered from our Customer Services team.

## Subscriptions

Our range of subscription services are designed to make using standards easier for you. For further information on our subscription products go to [bsigroup.com/subscriptions](http://bsigroup.com/subscriptions).

With **British Standards Online (BSOL)** you'll have instant access to over 55,000 British and adopted European and international standards from your desktop. It's available 24/7 and is refreshed daily so you'll always be up to date.

You can keep in touch with standards developments and receive substantial discounts on the purchase price of standards, both in single copy and subscription format, by becoming a **BSI Subscribing Member**.

**PLUS** is an updating service exclusive to BSI Subscribing Members. You will automatically receive the latest hard copy of your standards when they're revised or replaced.

To find out more about becoming a BSI Subscribing Member and the benefits of membership, please visit [bsigroup.com/shop](http://bsigroup.com/shop).

With a **Multi-User Network Licence (MUNL)** you are able to host standards publications on your intranet. Licences can cover as few or as many users as you wish. With updates supplied as soon as they're available, you can be sure your documentation is current. For further information, email [bsmusales@bsigroup.com](mailto:bsmusales@bsigroup.com).

## BSI Group Headquarters

389 Chiswick High Road London W4 4AL UK

## Revisions

Our British Standards and other publications are updated by amendment or revision.

We continually improve the quality of our products and services to benefit your business. If you find an inaccuracy or ambiguity within a British Standard or other BSI publication please inform the Knowledge Centre.

## Copyright

All the data, software and documentation set out in all British Standards and other BSI publications are the property of and copyrighted by BSI, or some person or entity that owns copyright in the information used (such as the international standardization bodies) and has formally licensed such information to BSI for commercial publication and use. Except as permitted under the Copyright, Designs and Patents Act 1988 no extract may be reproduced, stored in a retrieval system or transmitted in any form or by any means – electronic, photocopying, recording or otherwise – without prior written permission from BSI. Details and advice can be obtained from the Copyright & Licensing Department.

## Useful Contacts:

### Customer Services

**Tel:** +44 845 086 9001

**Email (orders):** [orders@bsigroup.com](mailto:orders@bsigroup.com)

**Email (enquiries):** [cservices@bsigroup.com](mailto:cservices@bsigroup.com)

### Subscriptions

**Tel:** +44 845 086 9001

**Email:** [subscriptions@bsigroup.com](mailto:subscriptions@bsigroup.com)

### Knowledge Centre

**Tel:** +44 20 8996 7004

**Email:** [knowledgecentre@bsigroup.com](mailto:knowledgecentre@bsigroup.com)

### Copyright & Licensing

**Tel:** +44 20 8996 7070

**Email:** [copyright@bsigroup.com](mailto:copyright@bsigroup.com)



...making excellence a habit.™