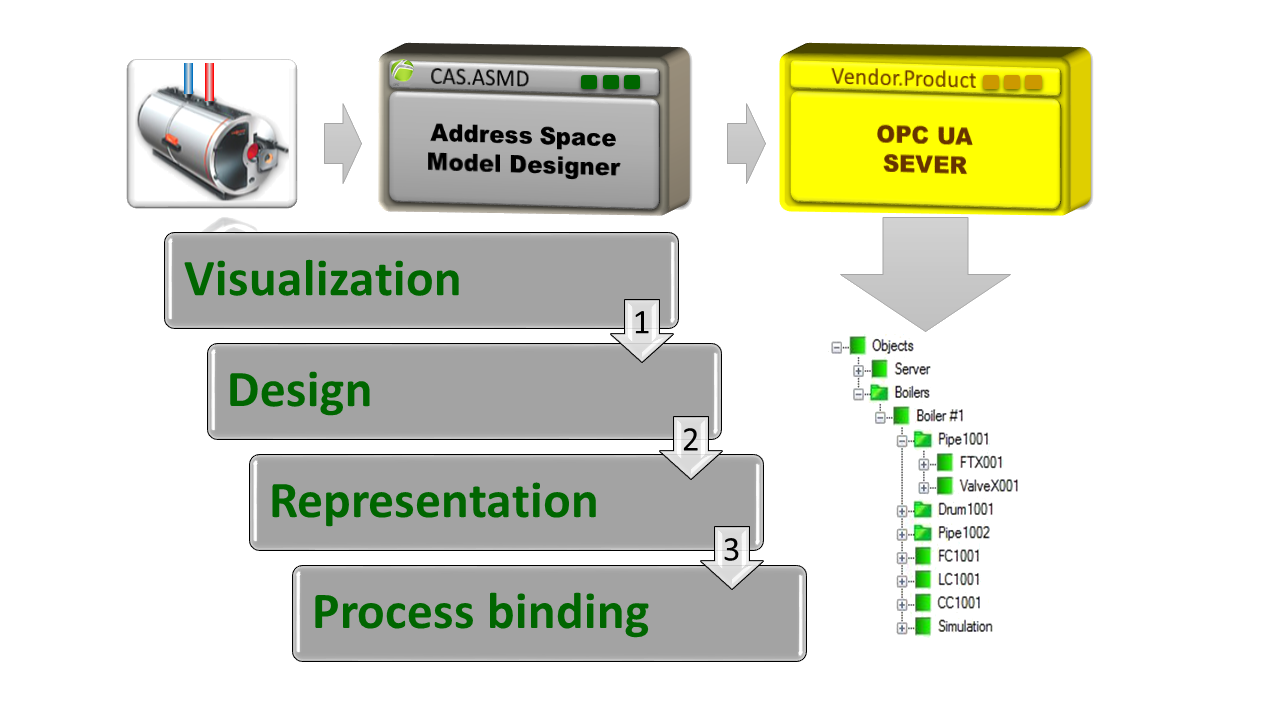


Manual

Address Space Interchange XML



Copyright CAS 2016

All rights reserved. No part of this document may be reproduced, stored in a retrieval system, translated, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of CAS.

|  |  |
| --- | --- |
| Product | Address Space Model Designer |
| release | 1.1 |
| Document |  |
| Date | 13 April 2016 |
| Authors | Mariusz Postół, PhD. Eng. (Project Manager) |
| Company | CAS  Poland 90-527 Łódź, ul. Wólczańska 124/134  tel/fax: (42) 686 25 47; (42) 686 50 28  [mpostol@cas.eu](mailto:mpostol@cas.eu)  <http://www.cas.eu>  <http://www.commsvr.com> |

Table of Contents

[Overview 2](#_Toc417305323)

[Address Space Model 2](#_Toc417305324)

[Address Space Representation 2](#_Toc417305325)

[Conventions 3](#_Toc417305326)

[UANodeSet 3](#_Toc417305327)

[UAVariable and UAVariableType 5](#_Toc417305328)

[Introduction 5](#_Toc417305329)

[XML representation 6](#_Toc417305330)

[Value 6](#_Toc417305331)

[DataType 6](#_Toc417305332)

[ValueRank 6](#_Toc417305333)

[ArrayDimensions 7](#_Toc417305334)

[TranslationType 7](#_Toc417305335)

[AccessLevel 8](#_Toc417305336)

[UserAccessLevel 9](#_Toc417305337)

[MinimumSamplingInterval 10](#_Toc417305338)

[Historizing 10](#_Toc417305339)

[UAObject and UAObjectType 10](#_Toc417305340)

[UAMethod 10](#_Toc417305341)

[MethodDeclarationId 11](#_Toc417305342)

[Executable 11](#_Toc417305343)

[UserExecutable 11](#_Toc417305344)

[UADataType 11](#_Toc417305345)

[UAView 13](#_Toc417305346)

[UAReferenceType 14](#_Toc417305347)

[InverseName 15](#_Toc417305348)

[Symmetric 15](#_Toc417305349)

[Base types 16](#_Toc417305350)

[UAType 16](#_Toc417305351)

[UAInstance 16](#_Toc417305352)

[UANode 16](#_Toc417305353)

[NodeId 17](#_Toc417305354)

[BrowseName 17](#_Toc417305355)

[SymbolicName 18](#_Toc417305356)

[WriteMask 18](#_Toc417305357)

[UserWriteMask 18](#_Toc417305358)

[DisplayName 19](#_Toc417305359)

[Description 19](#_Toc417305360)

[Category 19](#_Toc417305361)

[Documentation 19](#_Toc417305362)

[References 19](#_Toc417305363)

[Extensions 20](#_Toc417305364)

[EventNotifier 20](#_Toc417305365)

[Variant 20](#_Toc417305366)

[Data encoding 21](#_Toc417305367)

[General 21](#_Toc417305368)

[Overview 21](#_Toc417305369)

[Built-in Types 21](#_Toc417305370)

[Built-in Types Syntax 21](#_Toc417305371)

[Example 30](#_Toc417305372)

[UANodeSetChanges 32](#_Toc417305373)

Address Space Interchange XML

# Overview

## Address Space Model

Each OPC UA Server interchanges data with OPC UA Clients engaging the Address Space concept. More details about information modeling are captured in the whitepaper, which can be found here:

[OPC UA Information Model Deployment](http://www.commsvr.com/DownloadCenter/Publications/OPCUAInformationModelDeployment/tabid/563/language/en-US/Default.aspx)

Address Space instantiated by a server allows publication of the underlying process data in a standard way as a collection of NodeClass instances created according to the strictly observed rules defined by a not expandable Address Space Model. Each node in the Address Space is an instance of one of these NodeClasses.

As it was described in the article mentioned above the content of the particular Address Space is formally determined by a set of selected Information Models, which control:

* What Nodes are created;
* Nodes relationship (interconnections);
* Value of selected attributes of the Nodes.

Information Model is an expandable concept, hence the basic Information Model has been already expanded by some companion specifications and it is expected that this process will continue in the future as a result of adopting the standard to diverse application domains. Each Information Model is assigned a unique identifier to precisely distinguish a set of definitions contained in it. Any definition must be part of one Information Model only.

## Address Space Representation

To instantiate the Address Space according to a selected collection of Information Models a reusable OPC UA Server needs a description of it. The OPC UA standard defines a formal definition of the syntax and semantics of documents used for this purpose. An XML Schema is defined to describe syntax. The semantics of a compliant with this schema document is explained in this document.

The latest definition of the schema can be found at:

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

Generally speaking the document compliant with the schema provides two kinds of information:

* Address Space metadata – list of nodes, their relations and attribute values;
* Instruction how to create and maintain the Address Space.

The XML document compliant with the schema shall contain definitions of all the nodes including instance declarations defined by a selected Information Model. All the values must be evaluated as a result of inheritance chains processing. More can be found in an eBook available at:

<http://www.commsvr.com/UAModelDesigner/Index.aspx>

The XML document compliant with the schema may contain only a partial representation of a selected Address Space. For example this schema can be used to import or export an arbitrary set of nodes in the server Address Space. This partial representation can be used to save the server state for use by the server later on or to exchange with other applications, e.g. to:

* support offline configuration by a client,
* configure design tools.

## Conventions

For convenience the common “field” term is also used as reference to elements or attributes defined by the schema.

* NodeId: XML identifiers are emphasized using bold blue color.
* NodeId: Address Space Model terms emphasized using italic grey color.

# UANodeSet

The UANodeSet is the root element of the XML document. The fields of the element are shown in Table 1.

Table 1 UANodeSet

|  |  |  |
| --- | --- | --- |
| Field | XML Type | Description |
| NamespaceUris | UriTable | A list of namespaces represented by URI referred in the document. External references are allowed. |
| ServerUris | UriTable | A list of ServerUri entries used in the UANodeSet document. |
| Models | ModelTableEntry[] | A list of Models that are defined in the UANodeSet along with any dependencies these models have. |
| ModelUri | string | The URI for the model. This ModelUri should be one of the entries in the NamespaceUris table. |
| Version | string | The version of the model defined in the UANodeSet document. This is a human readable string and not intended for programmatic comparisons. |
| PublicationDate | dateTime | When the model was published. This value is used for comparisons if the Information Model is defined in multiple UANodeSet documents. |
| RequiredModels | ModelTableEntry[] | A list of dependencies for the model. If the model requires a minimum version the PublicationDate shall be specified. Tools which attempt to resolve these dependencies may accept any PublicationDate after this date. |
| Aliases | AliasTable | A list of aliases used in the UANodeSet document. |
| Extensions | xs:any | An element containing any vendor defined extensions to the UANodeSet. |
| LastModified | dateTime | The last time a document was modified. |
| <choice> | UAObject  UAVariable  UAMethod  UAView  UAObjectType  UAVariableType  UADataType  UAReferenceType | The Address Space nodes collection representation in the UANodeSet. |

The NamespaceUri is a list of URIs for namespaces referred to a set of selected OPC UA Information Models used by the UANodeSet. The NamespaceIndex used in NodeId, ExpandedNodeIds and QualifiedNames identifies an element in the list. The first index is always 1 (0 is always the OPC UA namespace). OPC UA namespace is not added to the table. At index 0 the table contains the namespace referred to the Information Model domain included in the XML document.

The ServerUris is a list of URIs for servers referenced in the UANodeSet. The ServerIndex in ExpandedNodeIds identifies an element in that list. The first index is always 1 (0 is always the current server).

The Mmdels element (added to the schema rel. 1.03) specifies the models, which are formally defined by the UANodeSet. It includes version information as well as information about any dependencies the model may have.

The Aliases are a list of string elements that can be used to substitute the numeric part (of type i) of NodeId (section NodeId). Aliases can be used to make the file more readable by allowing a string like ‘HasProperty’ to be used in place of a numeric NodeId (i=46). Aliases are optional.

UAObject, UAVariable, UAMethod, UAView are derived from UAInstance described in the section UAInstance. UAObjectType, UAVariableType, UADataType, UAReferenceType are derived from UAType described in the section UAType.

The Extensions enables to extend the XML document with elements not specified by the schema that can be used to attach custom vendor data to the UANodeSet.

# UAVariable and UAVariableType

## Introduction

In the Address Space Variable NodeClass is used to provide a value, which may be simple or complex. In the UANodeSet document this NodeClass is represented by UAVariable type. A UAVariable is a subtype of the UAInstance defined in UAInstance.

The type of the Variable is defined by VariableType NodeClass that is represented in the UANodeSet document by the UAVariableType. A UAVariableType is a subtype of the UAType defined in the section UAType.

Variables are always defined as Properties or DataVariables and are exposed as components of other nodes in the Address Space. They are never defined by themselves. A Variable node is always part of at least one other node, but may be related to any number of other nodes. These rules must be also observed in the UANodeSet document.

Properties are used to define the characteristics of nodes. Properties are instantiated using the Variable NodeClass.

Properties are the leaf of any hierarchy; therefore they shall not be the source node of any hierarchical references. This includes the references HasComponent or HasProperty types, that is, Properties do not contain Properties and, hence, cannot have complex structure. However, they may be the source node of any non-hierarchical references.

Properties shall always be defined in the context of another node and shall be the target node of at least one HasProperty reference. To distinguish them from DataVariables, they shall not be the target node of any reference of the HasComponent type. Thus, a reference of the HasComponent type pointing to a Variable node defines this node as a Property.

The BrowseName of a Property is always unique in the context of a node. Any node is not permitted to refer to two Variables using references of the HasProperty type having the same BrowseName.

DataVariables represent the content of an object. DataVariables are instantiated using the Variable NodeClass.

DataVariables identify their Properties using references of the HasProperty type. Complex DataVariables use HasComponent references to expose their component DataVariables.

DataVariables may use additional references to define relationships to other nodes. No restrictions are placed on the types of references used or on the NodeClasses of the nodes that may be referenced. However, restrictions may be defined by the ReferenceType excluding its use for DataVariables.

A DataVariable is intended to be defined in the context of an Object. However, complex DataVariables may expose other DataVariables, and ObjectTypes and complex VariableTypes may also contain DataVariables. Each DataVariable shall, therefore, be the target node of at least one HasComponent reference coming from an Object, an ObjectType, a DataVariable or a VariableType. DataVariables shall not be the TargetNode of any HasProperty references. Therefore, a HasComponent reference pointing to a Variable node identifies it as a DataVariable.

The reference of HasTypeDefinition type points to the VariableType used as type definition of the DataVariable.

If the DataVariable is used as instance declaration all nodes referenced with hierarchical references in the forward direction shall have unique BrowseNames in the context of this DataVariable.

## XML representation

The fields in the UAVariable and UAVariableType types are defined in Table 2.

Table 2 – UAVariable and UAVariableType fields

|  |  |  |
| --- | --- | --- |
| Field | XML Type | Description |
| Value | Variant | The Value of the node encoding using the UA XML wire encoding. |
| DataType | NodeId | The data type of the value. |
| ValueRank | ValueRank | The value rank. |
| ArrayDimensions | ArrayDimensions | The number of dimensions in an array value. |
| UAVariable only | | |
| Translation | TranslationType[] | A list of translations for the Value if the Value is a LocalizedText or a structure containing LocalizedTexts.  See the TranslationType section for more information. |
| 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. |

ArrayDimensions is ignored if ValueRank is not equal to the OneOrMoreDimensions.

### Value

The most recent value of the Variable node that the server has. Its data type is defined by the DataType field.

### DataType

NodeId of the DataType definition for the Value attribute of the represented NodeClass. The value of this field must be a valid NodeId (see the section NodeId). It is not required that the pointed out element is defined in the same NodeSet document. If that is the case many documents must be combined to resolve and validate this reference.

### ValueRank

This field indicates whether the Value attribute of the Variable NodeClass is an array and how many dimensions the array has.

It may have the following values:

* n > 1: the Value is an array with the specified number of dimensions.
* OneDimension (1): The value is an array with one dimension.
* OneOrMoreDimensions (0): The value is an array with one or more dimensions.
* Scalar (−1): The value is not an array.
* Any (−2): The value can be a scalar or an array with any number of dimensions.
* ScalarOrOneDimension (−3): The value can be a scalar or a one dimensional array.

NOTE: All DataTypes are considered to be scalar, even if they have array-like semantics like ByteString and String.

### ArrayDimensions

This field specifies the length of each dimension for an array value. It is intended to describe the capability of the Variable, not the current size.

The number of elements shall be equal to the value defined by the ValueRank. It shall be null if ValueRank ≤ 0. The value of 0 for an individual dimension indicates that the dimension has a variable length. For example, if a Variable is defined by the following C array:

Int32 myArray[346];

then the DataType would point to an Int32, the ValueRank has the value 1 and the ArrayDimensions is an array with one entry having the value 346.

Note that the maximum length of an array transferred on the wire is 2147483647 (max Int32) and a multidimensional array is encoded as a one dimensional array.

### TranslationType

This element applies only to UAVariable element and 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, there is one element in this array.

If the Value is a structure, each element contains translations for one or more fields identified by a name.

The TranslationType contains additional translations for LocalizedText entities used in the Value of the Variable NodeClass. The fields in the TranslationType are defined in Table 3. If multiple arguments exist there will be a Translation element for each argument.

The type can have two forms depending on whether the Value is a LocalizedText or a Structure DataType containing a LocalizedText list. 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 a unique name of a field 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, the name is an 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>

More information about the ExtensionObject element is captured in the section ExtensionObject.

Table 3 – 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. |

### AccessLevel

The AccessLevel attribute indicates the accessibility of the Value of a Variable node not taking user access rights into account and applies only to a UAVariable element. The AccessLevel attribute is used to indicate how the Value of a Variable node can be accessed (read/write) and if it contains current and/or historic data. The AccessLevel does not take any user access rights into account, i.e. although the Variable is writable this may be restricted to a certain user / user group.

The AccessLevel is an 8-bit unsigned integer with the structure defined in the Table 4.

Table 4 – AccessLevel bits meaning

|  |  |  |
| --- | --- | --- |
| Field | Bit | Description |
| CurrentRead | 0 | Indicates if the current value is readable; (0 means not readable, 1 means readable). |
| CurrentWrite | 1 | Indicates if the current value is writable (0 means not writable, 1 means writable). |
| HistoryRead | 2 | Indicates if the history of the value is readable (0 means not readable, 1 means readable). |
| HistoryWrite | 3 | Indicates if the history of the value is writable (0 means not writable, 1 means writable). |
| SemanticChange | 4 | Indicates if the Variable used as Property generates SemanticChangeEvents (see 9.33). |
| StatusWrite | 5 | Indicates if the current StatusCode of the value is writable (0 means not writable, 1 means writable). |
| TimestampWrite | 6 | Indicates if the current SourceTimestamp is writable (0 means not writable, 1 means writable). |
| Reserved | 7 | Reserved for future use. Shall always be zero. |

The first two bits also indicate if a current value of this Variable is available and the other two bits indicate if the history of the Variable is available in the OPC UA Server Address Space.

The SemanticChange bit of the AccessLevel shall be set when the Property describes the semantics of the node that owns the Property and changes of the Property value will generate SemanticChangeEvents. For example, a Property describing the engineering unit of a DataVariable will have a bit set, whereas a Property containing an Icon of the DataVariable will not. This behavior is exactly the same as described by the SemanticsChanged bit of the StatusCode. However, subscribing to a Variable one should look at the StatusCode to identify if the semantics has changed in order to receive this information before processing the value of the Variable.

### UserAccessLevel

The UserAccessLevel attribute is used to indicate how the Value attribute of a Variable NodeClass can be accessed (read/write) and if it contains current or historic data taking user access rights into account. It applies only to a UAVariable element. If the OPC UA Server does not have the ability to get any user access rights related information from the underlying system it should use the same bit mask as used in the AccessLevel attribute. The UserAccessLevel attribute can restrict the accessibility indicated by the AccessLevel, but not exceed it.

The UserAccessLevel is an 8-bit unsigned integer with the structure defined in the table Table 5.

Table 5 – UserAccessLevel bits meaning.

|  |  |  |
| --- | --- | --- |
| Field | Bit | Description |
| CurrentRead | 0 | Indicates if the current value is readable  (0 means not readable, 1 means readable). |
| CurrentWrite | 1 | Indicates if the current value is writable  (0 means not writable, 1 means writable). |
| HistoryRead | 2 | Indicates if the history of the value is readable  (0 means not readable, 1 means readable). |
| HistoryWrite | 3 | Indicates if the history of the value is writable (0 means not writable, 1 means writable). |
| Reserved | 4 | Reserved for future use. Shall always be zero. |
| StatusWrite | 5 | Indicates if the current StatusCode of the value is writable  (0 means not writable, 1 means writable). |
| TimestampWrite | 6 | Indicates if the current SourceTimestamp of the value is writable  (0 means not writable, 1 means writable). |
| Reserved | 7 | Reserved for future use. Shall always be zero. |

The first two bits also indicate if a current value of this Variable is available and the other two bits indicate if the history of the Variable is available via the OPC UA Server.

### MinimumSamplingInterval

The MinimumSamplingInterval attribute indicates how “current” the Value of the Variable NodeClass will be kept. It specifies (in milliseconds) how fast the server can reasonably sample the value for changes. The accuracy of this value (the ability of the server to attain “best case” performance) can be greatly affected by the system load and other factors.

A MinimumSamplingInterval of 0 indicates that the server is to monitor the item continuously. A MinimumSamplingInterval of -1 means indeterminate value.

### Historizing

The Historizing attribute indicates whether the server is actively collecting data for the history of the Variable node.

This differs from the AccessLevel which identifies if the Variable has any historical data. A value of TRUE indicates that the server is actively collecting data. A value of FALSE indicates the server is not actively collecting data. Default value is FALSE.

# UAObject and UAObjectType

*Objects* are used to represent systems, system components, real-world objects and software objects.

In the Address Space Object NodeClass is used to represent systems, system components, real-world objects and software objects. In the UANodeSet document this NodeClass is represented by a UAObject type. A UAObject is a subtype of the UAInstance defined in the section UAInstance.

In the Address Space the type of the Object NodeClass is defined by ObjectType NodeClass that is represented in the UANodeSet document by the UAObjectType. The UAObjectType type is a subtype of the UAType defined in the section UAType and does not extend this definition.

The UAObject defines an additional attribute, EventNotifier. The EventNotifier attribute is used to indicate if the node can be used to subscribe to events or the read/write historic events.

The EventNotifier is an 8-bit unsigned integer with the structure defined in Section EventNotifier.

# UAMethod

In the Address Space the Methods NodeClass defines callable functions. Methods are invoked remotely from the client using the call service. The invocations always run to completion and always return responses when complete.

Methods are represented in the UANodeSet document using the UAMethod type. UAMethod is a subtype of the UAInstance. It represents a Method NodeClass. The fields in the UAMethod type are defined in Table 6.

Table 6 – UAMethod

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| MethodDeclarationId | NodeId | The NodeId of a UAMethod in the subtype of the parent defiition. |
| Executable | boolean | Indicates whether the Method node is executable (False means not executable, True means executable) |
| UserExecutable | boolean | Indicates whether the Method node is executable (False means not executable, True means executable) taking user access rights into account. |

## **MethodDeclarationId**

This attribute may be specified for a Method node that is a target of a HasComponent reference from a single Object node. It is the NodeId of the UAMethod with the same BrowseName contained in the TypeDefinition associated with the Object node. If the TypeDefinition overrides a Method inherited from a base ObjectType,this attribute shall reference the Method node in the subtype.

## **Executable**

The Executable attribute indicates whether the Method node is executable (“False” means not executable, “True” means executable), not taking user access rights into account. If the server cannot get the executable information from the underlying system, it should state that it is executable. If a Method is called, the server should transfer this request and return the corresponding StatusCode if such a request is rejected.

## UserExecutable

The UserExecutable attribute indicates if the Method is currently executable taking user access rights into account (“False” means not executable, “True” means executable).

# UADataType

In the Address Space the DataType node is used to define simple and structured data types. Data types are used to describe the structure of the Value attribute of Variables and their VariableTypes. Each Variable and VariableType is, therefore, pointing with its DataType attribute to a node of the DataType NodeClass.

Nodes of the DataType NodeClass are represented in the USNodeSet document using the UADataType element. A UADataType is a subtype of the UAType defined in the section UAType. The fields in the UADataType type are defined in Table 7.

Table 7 – UADataType

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| Definition | DataTypeDefinition | A definition of the data type. |

The Definition element is a definition of the type that can be used by design tools to create code that can serialize any compatible value using OPC UA XML and/or OPC UA Binary encodings. It does not appear in the Address Space. This is only used to define subtypes of the Structure or Enumeration standard DataTypes.

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 8.

Table 8 – DataTypeDefinition

|  |  |  |
| --- | --- | --- |
| Field | XML Type | Description |
| Name | QualifiedName | A unique name for the data type encoded usin the syntax described in the section QualifiedName.  This field is only specified for nested DataTypeDefinitions.  The BrowseName of the DataType node is used otherwise. |
| SymbolicName | SymbolicName | 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 Name cannot be used for this purpose.  Only letters, digits or the underscore (‘\_’) are permitted.  This field is only specified for nested DataTypeDefinitions.  The SymbolicName of the DataType node is used otherwise. |
| BaseType | QualifiedName | The name of any base type.  Note that the BaseType can refer to types defined in other files.  The NamespaceUri associated with the Name should indicate where to look for the BaseType definition.  This field is only specified for nested DataTypeDefinitions.  The HasSubtype reference of the DataType NodeClass is used otherwise. |
| IsUnion | Boolean | This flag indicates if the data type represents a union.  Only one of the Fields defined for the data type is encoded into a value.  This field is optional. The default value is false.  If this value is true the first field is the switch value. |
| Field | 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. |

The Field element of the DataTypeField type 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 9.

Table 9 – DataTypeField

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| Name | string | A name for the field that is unique within the DataTypeDefinition. |
| SymbolicName | string | A symbolic name for the field that can be used in autogenerated code.  It should only be specified if the Name cannot be used for this purpose.  Only letters, digits or the underscore (‘\_’) are permitted. |
| DataType | NodeId | The NodeId of the DataType for the field.  This NodeId can refer to another Node with its own DataTypeDefinition.  This field is not specified for subtypes of Enumeration. |
| ValueRank | ValueRank | The value rank for the field.  It shall be Scalar (-1) or a fixed rank Array (>=1).  This field is not specified for subtypes of Enumeration. |
| 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 DataType Node for each structure.  This field is not specified for subtypes of Enumeration. |
| Value | int | The value associated with the field.  This field is only specified for subtypes of Enumeration. |
| IsOptional | boolean | The field indicates if a data type field in a structure is optional.  This field is optional. The default value is false.  This field is not specified for subtypes of Enumeration and Union. |

# UAView

Underlying systems are often large and clients could have an interest in only a specific subset of the data. They do not need, or want, to be burdened with viewing nodes in the Address Space for which they have no interest.

To address this problem, the OPC UA standard defines the concept of a view. Each view defines a subset of the nodes in the Address Space. The entire Address Space is the default view. Each node in a view may contain only a subset of its references, as defined by the creator of the view. The View node acts as the root for the nodes in the view. Views are defined using the View NodeClass, and are represented in the UANodeSet document by the UAView type. This type is specified in Table 10.

All nodes contained in a view shall be accessible starting from the root View node when browsing in the context of the view. It is not expected that all contained nodes can be browsed directly from the View node but rather browsed from other nodes contained in the view.

A View node may not only be used as an additional entry point into the Address Space but also as a construct to organize the Address Space and thus as the only entry point into a subset of the Address Space. Therefore, clients shall not ignore view nodes when browsing the Address Space. Simple clients that do not deal with views for filtering purposes can, for example, handle a View node like an object of the type FolderType.

Table 10 – UAView

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| ContainsNoLoops | boolean | Indication if following the references in the context of the view there are no loops. |
| EventNotifier | EventNotifier | Indication if the node can be used to subscribe to events or the read / write historic events. |

The ContainsNoLoops attribute is set to “true” to indicate that by following the References in the context of the view there are no loops, i.e. starting from node “A” contained in the view and following the forward references in the context of the view node “A” will not be reached again. It does not specify that there is only one path starting from the view node to reach a node contained in the view.

If set to “false” this attribute indicates that following references in the context of the view may lead to loops.

The EventNotifier is used to indicate if the node can be used to subscribe to events or the read / write historic events. The EventNotifier is an 8-bit unsigned integer. A detailed description of the EventNotifier type encoding is covered in the section EventNotifier.

# UAReferenceType

References are defined as instances of ReferenceType NodeClasses. ReferenceTypes are visible in the Address Space and are defined using the ReferenceType NodeClass. By contrast, a reference is an inherent part of a node and no NodeClass is used to represent references.

The OPC UA standard predefines a set of ReferenceTypes provided as an inherent part of the OPC UA Information Model. Servers may define custom ReferenceTypes. In addition, the OPC UA standard defines node management services that allow clients to add ReferenceTypes to the Address Space.

The inherited BrowseName attribute is used to specify the meaning of the ReferenceType as seen from the source node. For example, the ReferenceType with the BrowseName “Contains” is used in References which specify that the source node contains the target node. The inherited DisplayName attribute contains a translation of the BrowseName.

The BrowseName of a ReferenceType shall be unique in a server. It is not allowed that two different ReferenceTypes have the same BrowseName.

The IsAbstract attribute indicates if the ReferenceType is abstract. Abstract ReferenceTypes cannot be instantiated and are used only for organizational reasons, for example to specify some general semantics or constraints that its subtypes inherit.

A ReferenceType can have constraints restricting its use. For example, it can specify that starting from node A and only following References of this ReferenceType or one of its subtypes, it shall never be able to return to A, that is, a “No Loop” constraint.

The OPC UA standard does not specify how those constraints could or should be made available in the Address Space. Nevertheless, for the ReferenceTypes defined by the standard, some constraints are specified. This standard does not restrict the kind of constraints valid for a ReferenceType. It can, for example, also affect an ObjectType. The restriction that a ReferenceType can only be used by relating nodes of some NodeClasses with a defined cardinality is a special constraint of a ReferenceType.

ReferenceType NodeClasses is represented by the UAReferenceType in the UANodeSet document. A UAReferenceType is a subtype of the UAType defined in the section UAType. The fields in the UAMethod type are defined in Table 6.

Table 11 - UAReferenceType

|  |  |  |
| --- | --- | --- |
| Element | XML Type | Description |
| InverseName | LocalizedText | The inverse name of the Reference, that is the meaning of the ReferenceType as seen from the TargetNode |
| Symmetric | boolean | The value is true if the meaning of the ReferenceType is the same as seen from both the source node and the target node, otherwise false. |

## InverseName

InverseName attribute of the LocalizedText type defines inverse name of the reference that is the meaning of the ReferenceType as seen from the target node.

If a ReferenceType is symmetric, the InverseName attribute shall be omitted. Examples of symmetric ReferenceTypes are “Connects To” and “Communicates With”. Both imply the same semantics coming from the source node or the target node. Both directions are, therefore, considered to be forward references.

If the ReferenceType is non-symmetric and not abstract, the InverseName attribute shall be set. The InverseName specifies the meaning of the ReferenceType as seen from the target node. Examples of non-symmetric ReferenceTypes include “Contains” and “Contained In”, and “Receives From” and “Sends To”.

References that use the InverseName, such as “Contained In” References, are referred to as inverse References.

As an example of a unidirectional reference, it is often the case that a subscriber knows its publisher, but the publisher does not know its subscribers. The subscriber would have a “Subscribes To” Reference to the publisher, without the publisher having the corresponding “Publishes To” inverse References to its subscribers.

The DisplayName attribute of the UAType and the InverseName are the only standardized places to indicate the semantics of a ReferenceType. There may be more complex semantics associated with a ReferenceType than can be expressed in those attributes (e.g. the semantics of HasSubtype). The OPC UA specification does not define how this semantics should be exposed. However, the Description attribute of the ReferenceType NodeClass can be used for this purpose.

## Symmetric

The Symmetric attribute is used to indicate whether or not the meaning of the represented ReferenceType is the same for both the source node and target node. Symmetric attribute of the boolean has the following values:

* TRUE - the meaning of the ReferenceType is the same as seen from both the source node and the target node.
* FALSE - the meaning of the ReferenceType as seen from the target node is the inverse of that seen from the source node.

# Base types

## UAType

A UAType is a subtype of the UANode defined in the section UANode. It is the base type for the XML types listed in Table 12.

Table 12 - UANodeSet Subtypes

|  |  |
| --- | --- |
| Subtype | Description |
| UAObjectType | Defines an ObjectType Node as described in ... |
| UAVariableType | Defines a VariableType Node as described ... |
| UADataType | Defines a DataType Node as described ... |
| UAReferenceType | Defines a ReferenceType Node as described... |

## UAInstance

A UAInstance is a subtype of the UANode defined in the section UANode. It is the base type for the types defined in Table 13.

Table 13 - UAInstance Subtypes

|  |  |
| --- | --- |
| Subtype | Description |
| UAObject | Defines an Object node as described in... |
| UAVariable | Defines a Variable node as described in... |
| UAMethod | Defines a Variable node as described.. |
| UAView | Defines a View node as described in... |

The fields in the UAInstance type are defined in Table 14.

Table 14 – UAInstance

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| ParentNodeId | NodeId | The NodeId of the parent. |

ParentNodeId of NodeId type of the node contains a value pointing to the parent node within the Address Space portion represented by the UANodeSet document. This field is used to indicate that a tight coupling exists between the node and its parent (e.g. when the parent is deleted the child is deleted as well). This information is not exposed in the Address Space, i.e. it does not appear in the Address Space and is intended for use by design tools.

## UANode

The OPC UA Address Space Model defines a BaseNodeClass from which all other NodeClasses are derived. The derived NodeClasses represent various components of the OPC UA Object Model. A UANode XML type is an abstract base type that represents the Base NodeClasse attributes. It defines a set of fields representing attributes and references of any node in the Address Space. There are the following XML subtypes: UAObject, UAVariable, UAMethod, UAView, UAObjectType, UAVariableType, UADataType, UAReferenceType; each one for an appropriate NodeClass defined in the UANodeSet schema. Each of these subtypes defines XML fields for the OPC UA attributes specific to the selected NodeClass. The fields in the UANode type are defined in Table 15.

Table 15 – UANode Type

|  |  |  |
| --- | --- | --- |
| Field | XML Type | Description |
| NodeId | NodeId | It holds the value of the NodeId attribute of modes in the Address Space. |
| BrowseName | QualifiedName | It holds the value of the BrowseName attribute of modes in the Address Space . |
| SymbolicName | string | A symbolic name for the node. |
| WriteMask | WriteMask | The value of the WriteMask node attribute. |
| UserWriteMask | WriteMask | The value of the UserWriteMask node attribute . |
| DisplayName | LocalizedText[] | A list of DisplayNames for the node in different locales . |
| Description | LocalizedText[] | A list of the Description entries for the Node in different locales. |
| Category | string[] | Grouping information necessary in UANodeSet |
| Documentation | string | Non-localized text comments and documentation. |
| References | Reference[] | A list of References for the node. |
| Extensions | xs:any | An element containing any vendor defined extensions to the UANode. |

### NodeId

Nodes are unambiguously identified using a constructed identifier called the NodeId. Some servers may accept alternative NodeIds in addition to the canonical NodeId represented in this attribute. A server shall preserve the value of the NodeId *attribute* of a node, that is, it shall not generate new values when rebooting.

### BrowseName

Nodes have a BrowseName *a*ttribute that is used as a non-localized human-readable name when browsing the Address Space to create paths out of BrowseNames. The TranslateBrowsePathsToNodeIds service can be used to follow a path constructed of BrowseNames.

A BrowseName should never be used to display the name of a node. The DisplayName should be used instead for this purpose.

Unlike NodeId, the BrowseName cannot be used to unambiguously identify a node. Different nodes may have the same BrowseName.

The section BrowseName defines the structure of the BrowseName. It contains a namespace and a string. The namespace is provided to make the BrowseName unique in some cases in the context of a Node (e.g. Properties of a Node) although not unique in the context of a server Address Space. If different organizations define BrowseNames for Properties, the namespace of the BrowseName provided by the organization makes the BrowseName unique, although different organizations may use the same string having a slightly different meaning.

Servers may often choose to use the same namespace for the NodeId and the BrowseName. However, if they want to provide a standard Property, itsBrowseName shall have the namespace of the standard body although the namespace of the NodeId reflects something else, for example the local server.

It is recommended that standard bodies defining standard type use their namespace for the NodeId of the TypeDefinitionNode as well as for the BrowseName of the TypeDefinitionNode*.*

The string-part of the BrowseName is case sensitive. That is, Clients shall consider it case sensitive. Servers are allowed to handle BrowseNames passed in service requests as case insensitive. Examples are the TranslateBrowsePathsToNodeIds service or Event filter.

### SymbolicName

A symbolic name for the node that can be used as a class/field name by a design tools to enhance auto-generated code. It should only be specified if the BrowseName cannot be used for this purpose. This field is not used directly to instantiate Address Space and is intended for use by design tools. Only letters, digits or the underscore (‘\_’) are permitted. This attribute is not exposed in the Address Space.

### WriteMask

The optional WriteMask attribute represents the WriteMask attribute of the Basic NodeClass, which exposes the possibilities of a client to write the attributes of the node. The WriteMask attribute does not take any user access rights into account, that is, although an attribute is writable this may be restricted to a certain user/user group.

If the OPC UA Server does not have the ability to get the WriteMask information for a specific attribute from the underlying system, it should state that it is writable. If a write operation is called on the attribute, the server should transfer this request and return the corresponding status code if such a request is rejected.

The WriteMask *a*ttribute is a 32-bit unsigned integer with the structure defined in Table 23. If the bit is set to 0, it means the attribute is not writable, if it is set to 1, it means it is writable. If a node does not support a specific attribute, the corresponding bit has to be set to 0.

Table 16 – Bit mask for WriteMask and UserWriteMask

|  |  |  |
| --- | --- | --- |
| Field | Bit | Description |
| AccessLevel | 0 | Indicates if the AccessLevel Attribute is writable. |
| ArrayDimensions | 1 | Indicates if the ArrayDimensions Attribute is writable. |
| BrowseName | 2 | Indicates if the BrowseName Attribute is writable. |
| ContainsNoLoops | 3 | Indicates if the ContainsNoLoops Attribute is writable. |
| DataType | 4 | Indicates if the DataType Attribute is writable. |
| Description | 5 | Indicates if the Description Attribute is writable. |
| DisplayName | 6 | Indicates if the DisplayName Attribute is writable. |
| EventNotifier | 7 | Indicates if the EventNotifier Attribute is writable. |
| Executable | 8 | Indicates if the Executable Attribute is writable. |
| Historizing | 9 | Indicates if the Historizing Attribute is writable. |
| InverseName | 10 | Indicates if the InverseName Attribute is writable. |
| IsAbstract | 11 | Indicates if the IsAbstract Attribute is writable. |
| MinimumSamplingInterval | 12 | Indicates if the MinimumSamplingInterval Attribute is writable. |
| NodeClass | 13 | Indicates if the NodeClass Attribute is writable. |
| NodeId | 14 | Indicates if the NodeId Attribute is writable. |
| Symmetric | 15 | Indicates if the Symmetric Attribute is writable. |
| UserAccessLevel | 16 | Indicates if the UserAccessLevel Attribute is writable. |
| UserExecutable | 17 | Indicates if the UserExecutable Attribute is writable. |
| UserWriteMask | 18 | Indicates if the UserWriteMask Attribute is writable. |
| ValueRank | 19 | Indicates if the ValueRank Attribute is writable. |
| WriteMask | 20 | Indicates if the WriteMask Attribute is writable. |
| ValueForVariableType | 21 | Indicates if the Value Attribute is writable for a VariableType. It does not apply for Variables since this is handled by the AccessLevel and UserAccessLevel Attributes for the Variable. For Variables this bit shall be set to 0. |
| Reserved | 22:31 | Reserved for future use. Shall always be zero. |

### UserWriteMask

The optional UserWriteMask attribute holds the value of the UserWriteMask attribute of the Basic NodeClass, which exposes the possibilities of a client to write the attributes of the node taking user access rights into account. It uses the same bit mask as used in the WriteMask attribute defined in Table 23.

The UserWriteMask attribute can only further restrict the WriteMask attribute*,* when it is set to not writable in the general case that applies for every user.

### DisplayName

The DisplayName *a*ttribute contains the localized name of the node. Clients should use this attribute if they want to display the name of the node to the user. They should not use the BrowseName for this purpose. The server may maintain one or more localized representations for each DisplayName. Clients negotiate the locale to be returned when they open a session with the server. The section DisplayName defines the structure of the DisplayName. The string part of the DisplayName is restricted to 512 characters.

### Description

The optional Descriptionelement shall explain the meaning of the node in a localized text using the same mechanisms for localization as described for the DisplayName.

### Category

The Category element of the string list type is a collection of identifiers used to group related UANodes together for use by design tools that create/edit UANodeSet files. This attribute is not exposed in the Address Space.

### Documentation

Additional non-localized documentation for use by tools that create/edit UANodeSet files. It provides a mechanism for designers to document their document. This attribute is not exposed in the Address Space.

### References

This collection represents all the references defined by the selected Information Model including references to the instance declarations nodes. The References list specifies references that must be created for the node during Address Space instantiation. The reference can be forward or inverse. Only one direction for each reference needs to be specified in the UANodeSet document. The other direction shall be added automatically during any import operation. The fields in the Reference type are defined in Table 17.

Table 17– Reference

|  |  |  |
| --- | --- | --- |
| Field | XML Type | Description |
| value | NodeId | The target node identifier. |
| ReferenceType | NodeId | The node identifier providing the type definition of this reference. |
| IsForward | boolean | true if this node is the source node, otherwise false. |

The value of the Reference element holds the value of NodeId type of the target node. The syntax of the NodeId value is defined in …. (ExpandedNodeId). This value can be replaced by an alias.

ReferenceType of the NodeId type represents a dependency on the selected ReferenceType node. The syntax of the value that is compatible with the NodeId type is defined in the section NodeId. This value can be replaced by an alias.

### Extensions

The Extensions enables to extend the XML document with elements not specified by the schema that can be used to attach vendor defined data to the UANode.

A UANodeSet is expected to contain many UANode elements which reference one another. Tools that create a file compliant with the UANodeSet schema should not add Reference elements for both directions in order to minimize the size of the XML file. Tools that read the XML file shall automatically add reverse references unless reverse references are not appropriate taking into consideration the Symmetric field value. HasTypeDefinition and HasModellingRule defined by the OPC UA Information Model are two examples where it is not appropriate to add reverse references.

## EventNotifier

The EventNotifier represents the mandatory EventNotifier *attribute of the* ObjectNodeClass and identifies whether the o*bject* can be used to subscribe to e*vents* or to read and write the history of the e*vents*. The definition of this type is captured in the Table 18.

Table 18 - EventNotifier

|  |  |  |
| --- | --- | --- |
| Field | Bit | Description |
| SubscribeToEvents | 0 | Indicates if it can be used to subscribe to *Events*  (0 means cannot be used to subscribe to *Events*, 1 means can be used to subscribe to *Events*). |
| Reserved | 1 | Reserved for future use. Shall always be zero. |
| HistoryRead | 2 | Indicates if the history of the *Events* is readable  (0 means not readable, 1 means readable). |
| HistoryWrite | 3 | Indicates if the history of the *Events* is writable  (0 means not writable, 1 means writable). |
| Reserved | 4:7 | Reserved for future use. Shall always be zero. |

## Variant

The Variant data type syntax described in the section …. Is used to encode the value for the Value of UAVariable or UAVariableType elements. This syntax is the same as used over the wire by the services. 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 EncodingId and a Structure with fields could be NodeIds, ExpandedNodeIds or QualifiedNames. The NamespaceIndexes and ServerIndexes in these fields shall also reference the tables in the UANodeSet.

# Data encoding

## General

### Overview

Instead of definition custom types the data (i.e. values of the selected fields in the UANodeSet XML document) is encoded using OPC UA XML syntax. This section is focused on data encoding, which describes lexical grammar, i.e. how to construct values in the document.

### Built-in Types

OPC UA XML d*ata encoding* is based on rules that are defined for a standard set of the built-in data types. These built-in types are then used to construct structures, arrays and messages. The built-in types are described in Table 19.

Table 19 – 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 255. |
| 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 4 294 967 295. |
| 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 (custom) 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. |

### Built-in Types Syntax

#### 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 2: XML Schema Part 2: Datatypes](http://www.w3.org/TR/xmlschema-2/)

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

#### Boolean

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

#### 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 20.

Table 20 – 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 |

#### 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 21.

Table 21 – 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).

#### String

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

#### 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 are represented in UTC format.

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

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

The latest date/time value on a development platform shall be encoded in XML as '9999-12-31T23:59:59Z'

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

The earliest date/time value on a development platform is equivalent to a null date/time value.

#### Guid

A Guid is a 16-byte globally unique identifier with the syntax defined in Table 22.

Table 22 – 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

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>

#### 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 development platforms will not be able to preserve the distinction between a ByteString and a one dimensional array of Byte.

If a decoderfor a development platform cannot preserve the distinction it shall convert all one dimensional arrays of Byte to ByteStrings value.

Each element in a one dimensional array of ByteString can have a different length which means it is structurally different from a two dimensional array of bytewhere 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 development platform does not supportunsigned integers, it will have to represent ByteStrings as arrays of SByte. In this case, the requirements for Byte would then apply to SByte.

A ByteString value is encoded as an xs:base64Binary value. The XML schema for a ByteStringis:

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

#### 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>

XmlElements may only be used inside Variant or ExtensionObject values.

#### NodeId

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

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

The elements of this syntax are described in Table 23.

Table 23 – NodeId syntax

|  |  |  |
| --- | --- | --- |
| Field | XML Type | Description |
| <namespaceindex> | UInt16 | The namespaceIndex is encoded 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 type of the <value> part.  The flag has the following values:   |  |  | | --- | --- | | i | NUMERIC (UInteger) | | s | STRING (String) | | g | GUID (Guid) | | b | OPAQUE (ByteString) | |
| <value> | \* | The *Identifier* encoded as string.  The *Identifier* is formatted using the XML data type mapping for the <type>.  Note that the *Identifier* 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 describing syntax of the NodeId value is as follows:

<xs:complexType name="NodeId">

<xs:sequence>

<xs:element name="Identifier" type="xs:string" minOccurs="0" />

</xs:sequence>

</xs:complexType>

#### ExpandedNodeId

An ExpandedNodeIdvalue 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 24.

Table 24 – Components of ExpandedNodeId

|  |  |  |
| --- | --- | --- |
| Field | Data Type | Description |
| <serverindex> | UInt32 | The <serverindex> *is* formatted as a base 10 number.  If the <serverindex> is 0 then the entire 'svr=0;' clause shall be omitted. |
| <namespaceindex> | UInt16 | The <namespaceIndex> formatted as a base 10 number.  If the <namespaceIndex> is 0 then the entire 'ns=0;' clause shall be omitted.  The <namespaceIndex> shall not be present if the URI is present. |
| <uri> | String | The URI 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 *URI* is null or empty then 'nsu=;' clause shall be omitted. |
| <type> | Enum | A flag that specifies the *IdentifierType.*  This field is described in  Table 23. |
| <value> | \* | The *Identifier* encoded as string.  This field is described in  Table 23. |

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>

#### StatusCode

A StatusCodeis 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>

#### DiagnosticInfo

A DiagnosticInfovalue 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="Locale" type="xs:int" minOccurs="0/>

<xs:element name="LocalizedText" 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>

#### QualifiedName

A QualifiedName is represented as a string with the syntax:

<NamespaceIndex>:<name>

where the NamespaceIndex refers to the NamespaceUris table.

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>

#### 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>

#### ExtensionObject

An ExtensionObjectis a container for any value of a custom structured data type which cannot be encoded as one of the other built-in data types. The ExtensionObjectcontains a complex value serialized as an XML element. It also contains an identifier which indicates what data it contains and how it is encoded.

Structured data types are represented in a server Address Space as sub-types of the Structure DataType defined by the standard OPC UA Information Model. The data encodings available for any given structured DataType are represented as an Object *of* DataTypeEncodingType in the *server* Address Space. The NodeId attribute for this Object is the identifier stored in the ExtensionObject.

*Server* implementers should use namespace qualified numeric NodeIds for any Object *of* DataTypeEncodingType they define. This will minimize the overhead introduced by packing structured values into an ExtensionObject.

An ExtensionObjectvalue 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 ExtensionObjectcontains 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 of NodeId type for the Object of DataTypeEncodingType.

#### 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 with a data type derived from BaseDataType.

Variants can be empty. An empty Variant is described as having a null value and should be treated like a null column in an 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* DiagnosticInfotypes only have meaning when returned in a response message with an associated status code. As a result,Variants cannot contain values of DataValue or DiagnosticInfo*.* This requirement means that if a node’s attribute supports the writing of a null value it shall also support writing of an empty Variantand vice versa.

The Value attribute of Variables with a DataType of BaseDataType is 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.

A Variantvalue 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 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 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 a sequence 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. All dimensions shall be specified and shall be greater than zero. If the dimensions are inconsistent with the number of elements in the array the decoder shall stop and report error.

The complete set of built-in type names is listed in Table 19.

#### DataValue

A value of DataValue type is encoded as an 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>

# Example

This example defines the nodes for an Address Space with the URI of:

“http://sample.com/Instances”.

This example references nodes defined in the base OPC UA Information Model and an Information Model with the URI:

“http://sample.com/Types”.

The XML namespaces declared by the root element 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 Information Model, i.e.:

"http://opcfoundation.org/UA/".

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 element defines new DataType node in the Address Space that inherits from a DataType defined in another Information Model (ns=2;i=314). It is assumed that any application importing this file will already know about the referenced Information Model. A server could map the references onto another OPC UA Server by adding a <ServerIndex> to identify the target node (described in section References). The structure of the DataType is defined by the Definition element. This information can be used by code generators to automatically create serializers for this DataType.

<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">

<Field Name="NoOfGears" DataType="UInt32" />

<Field Name="ManufacterName" DataType="QualifiedName" />

</Definition>

</UADataType>

The following XML element defines an Object node in the Address Space of type that is defined in another Information Model (ns=2;i=341). It has a single Property (ns=1;i=376) 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>

The following XML element defines the **PrimaryVehicle** Variable node of the PropertyType type that is defined in base OPC UA Information Model (i=68) and DataType pointing out to definition in another Information Model (ns=2;i=314). The referenced definition is also the base type for the BicycleType of the DataType NodeClass. The AccessLevels attribute declares the Variable node as Readable and Writeable (see section AccessLevel). The ParentNodeId indicates that this element represents a node that is tightly coupled with the parent node represented by the DriverOfTheMonth element. This information is only a request to delete this node if the parent is deleted. It is used only while the XML document is processed.

<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>

It is worth stressing that the namespace index of the NodeId and BrowseName attributes are different. It is the result of different contexts (namespaces) of the type of the node definition and the node instantiation.

The following Value element represents an example of the value 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 ManufacterName field refers to the NamespaceUris table in this document. The application is responsible for changing whatever value it needs 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:ManufacterName>

<uax:NamespaceIndex>1</uax:NamespaceIndex>

<uax:Name>Hello</uax:Name>

</s1:ManufacterName>

</s1:BicycleType>

</Body>

</ExtensionObject>

</Value>

</UAVariable>

These are the DataTypeEncoding nodes for the BicyleType 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 BicyleType 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 DataTypeDictionary node for the DataTypeDescription declared above. The XML Schema document is a UTF-8 document stored as xs:base64Binary value (see Base64). This allows Clients to read the schema for all DataTypes which belong to the DataTypeDictionary. The value of DataTypeDescription Node for each DataType contains an XPath query that will find the correct definition inside the schema document.

<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...W1hPg==</uax:ByteString>

</Value>

</UAVariable>

# UANodeSetChanges

The UANodeSetChanges is the root of a document that contains a set of changes to an Address Space. It is expected that a single file will contain either a UANodeSet or a UANodeSetChanges element at the root. It provides a list of nodes and references to be added and/or a list nodes and references to be deleted.