Matrikon® Flex OPC UA SDK

User's Manual

Table of Contents

Matrikon® Flex OPC UA SDK 1

Table of Contents 2

Table of Tables 10

Table of Figures 10

Copyright and Trademark Information 11

Document Revision History 13

Overview of Manual 15

References 15

Terms and Abbreviation 15

Introduction to OPC UA 17

OPC UA Applications 18

OPC UA Server Application 18

OPC UA Client Application 21

Security in OPC UA 22

Profiles 22

Introduction to Matrikon® Flex OPC UA SDK 23

SDK Specification 24

Supported Profiles 24

Supported Transport Protocol 25

Supported Security Facet 25

Limitation 25

Address Space 25

User Permissions 26

Data Source Integration *26*

Platform Requirement 26

CPU *26*

Compiler *26*

IEEE 754 Floating Point Conformance *26*

TCP/IP Stack *27*

Time Synchronization *27*

Runtime Environment *28*

Security in SDK 28

Secured Communication *28*

Security Considerations *28*

Integrating Security *28*

Certificate Stores *29*

Endpoint Configuration *29*

User Permissions *30*

Address Space 31

SDK Address Space Design *32*

Configuring SDK (Build Configuration) 32

UASDK Platform Layer Components *32*

Security Related Configurations *35*

Generic Build Macros *41*

Miscellaneous 42

Locales and Translations *42*

Memory Management *42*

Logging and Debug *43*

Services specific *44*

Data Access, Events and History *44*

Certificate Management *45*

SDK Distribution and Examples *45*

Creating an OPC UA Server 50

Creating Server 50

Configuring SDK *50*

XML Import *51*

Reading and Writing Variables *51*

Events, Alarms and Conditions *53*

Enable/Disable Server Application to Receive Event Notifications *55*

Callbacks/Listeners *55*

Filters *55*

Profiles *55*

Intervals *56*

Building Server on Different Platform 56

Memory Usage and Statistics and Performance 56

Memory Utilization *56*

FLASH and RAM *57*

Performance Issues *58*

Available Features in SDK 59

Attribute Changed Listener *59*

Attribute About To Be Read Listener *59*

Address Space Listener *60*

Server State Listener *60*

Service Call Listener *60*

Secure Channel Listener *61*

Miscellaneous 61

Registering Server to Discovery Server *62*

Server Diagnostics *62*

Address Space Roles and Permissions 62

Introduction *62*

Roles *62*

Permissions *63*

RolePermissions *63*

Assigning Roles *63*

Calculating Permissions *63*

RoleSet Information Model *63*

Enabling Roles and Permissions *64*

Role and Permissions Related Classes *64*

Node Management Service 65

Introduction *65*

Server-side implementation *65*

Add Nodes service *65*

Working *66*

Delete Nodes service *66*

Working *66*

Add References service *66*

Working *67*

Delete References service *67*

Working *67*

Server Access Type 67

Data Access *68*

Events, Alarm and Conditions *72*

History Access *72*

Creating OPC UA Client 73

Creating Client 73

Building Client on Different Platform 73

Available Features in SDK 73

Publish Subscribe 75

Introduction to Pub Sub 75

Entities *75*

The PubSubState State Machine *79*

Pub Sub Support in the FLEX SDK 80

Limitations and Considerations *80*

Platform Configuration *81*

PubSub Configuration *82*

Example configuration of a Publisher using the helper *86*

Example configuration of a Subscriber using the helper *90*

Creating Event Filters *95*

API References 97

Security APIs 97

Crypto Library APIs *101*

Certificate Management Helpers 113

CertMgmtPushHelper_t *114*

CertMgmtPullHelper_t *114*

Build Macros 117

Generic Build Macros *117*

Services specific *120*

Data Access, Events and History *121*

Certificate Management *122*

Logging and Debug *122*

Server Diagnostics *123*

Miscellaneous 123

Registering Server to Discovery Server *124*

Locales and Translations *124*

Complex Type Support 124

Introduction *124*

FLEX SDK complex type components *125*

Working with the FLEX SDK complex type classes *127*

Reverse Connect 136

Server 136

User Authentication APIs *136*

Runtime Configuration APIs *137*

Address Space *150*

Other Modules in Server 162

Access Types 163

Events, Alarms and Conditions APIs 163

Service Call Listener Interface 180

Client 181

IClientNotificationCallback_t 181

BasicClientNotificationManager_t 182

PublishResponseHandler_t 183

RepublishResponseHandler_t 184

RecoverableClientSession_t 185

RecoverableClientSession_t::IRecoverableClientSessionListener_t 186

IClientConfiguration_t 186

IClientCore_t 186

IClientMonitoredItem_t 198

IClientSession_t 201

IClientSession_t::ISessionListener_t 217

IClientChannel_t 218

IClientChannel_t::IChannelListener_t 218

IClientSubscription_t 219

IClientUserCertificate_t 228

IModifyClientMonitoredItemsComplete_t 228

ISetMonitoringModeComplete_t 229

ITcpConnectionInitiator_t 229

MonitoredItemOperationResult_t 230

AggregateType_t ... 231

Helper Classes for Reading and Updating Historical Data&Event ... 231

PubSub ... 233

PublishSubscribeFactory_t ... 233

IPublishSubscribe_t ... 233

DataSetFolder_t ... 235

PublishedDataSet_t ... 236

IPubSubConnection_t ... 237

IWriterGroup_t ... 239

IReaderGroup_t ... 240

IDataSetWriter_t ... 241

IDataSetReader_t ... 243

Appendix ... 246

Appendix A: Third-Party Licenses including Open Source Software ... 246

Appendix B: Assessing the amount of RAM required for an embedded UA Server ... 247

Introduction ... 247

Estimation technique ... 247

Configure the allocator ... 247

Configure the server ... 248

Stress the server to the limits of its configured capabilities ... 248

Read out the memory metrics ... 249

Table of Tables

Table 1 - Terms and Definitions........ 16

Table 2 – Memory Management Guidelines 43

Table 3 - Minimum Server Footprint 58

Table 4 – Typical Server Performance 59

Table 5 – Third-Party Components used in SDK 247

Table of Figures

Figure 1 – OPC UA communication 17

Figure 2 – OPC UA Server 18

Figure 3 – OPC UA Client 21

Figure 4 – OPC UA Cross-Platform Application 23

Figure 5 – OPC UA Server and Client 24

Figure 6 - Publisher Data Flow 76

Figure 7 - Subscriber Data Flow 78

Figure 8 - The components containing a PubSubState state machine and their relationships 79

Figure 9 - PubSubState state machine 80

Copyright and Trademark Information

SOFTWARE VERSION

Version: R5.0

DOCUMENT VERSION

Version: 5.01

COPYRIGHT INFORMATION

CONFIDENTIAL

LIMITATIONS

Document Revision History

Date	Document Version	Description	Author
2016-09-09	1.0	Initial revision	LP
2016-09-30	2.0	Copyright information updated with latest	BV
2016-11-30	2.5	Base Events updates	MA
2017-03-24	2.8	Appendix section Updates	DP
2017-03-29	3.0	Edits for R400.1 release	LP
2017-03-30	4.0	Edits for R400.1 release package	RJ
2017-04-26	4.1	New title graphic	LP
2017-06-01	4.2	Updated styles and standards for the entire document	DP
2017-07-04	4.3	Updated position of Title, Copyright, Revision and TOC, Language voice, warning note for certification section and Appendix	DP
2017-07-10	4.4	Reorganized document.	MA
2017-08-23	4.5	Edited document	DP
2017-11-08	4.6	Edited document	DP
2018-05-29	4.7	Edited for new supported features in SDK R410.1 release	BV
2018-10-10	4.8	Edited for new supported features in SDK R410.2 release	BV
2019-03-13	4.9	Edited for new supported features in SDK R410.3 release	BV / RJ

Date	Document Version	Description	Author
2019-12-11	4.10	Edited for new supported features in SDK R410.4 release	LP
2020-04-15	4.11	SDK Version number updated	LP
2021-02-02	5.00	Pub/Sub description initial draft	LP
2021-05-14	5.01	Pub/Sub description updated for GA release	LP

Overview of Manual

This document uses the following conventions to highlight valuable information. These conventions assist the user throughout the manual.

Element	Utility
⚠	This symbol denotes important information that must be acknowledged.
BOLD	Font displayed in this color and style indicate a hyperlink to the applicable/associated information within this document, or if applicable, any external sources.
Note:	Important information regarding a topic is emphasized with a **Note**.

The *User's Manual* has been authored in a way that the user clicks on references in the document to easily navigate to the referenced point without having to scroll through several pages (in some cases). For example, if the user were to see the sentence "*Refer to Figure 1 for more information*", pressing the **CTRL** key and clicking on the text "*Figure 1*" automatically takes the user to the location of Figure 1 within the document.

References

- **www.opcfoundation.org**
- **www.matrikonopc.com**
- **www.opcsupport.com**

Terms and Abbreviation

Term/Abbreviation	Description
CRL	Certificate Revocation List
VM	Virtual Machine
API	Application Program Interface
HMI	Human Machine Interface
SCADA	Supervisory Control and Data Acquisition
PLC	Programmable Logic Controller
SDK	Software Development Kit
TCP/IP	Transmission Control Protocol/Internet Protocol
DA	Data Access Types

Term/Abbreviation	Description
AC	Events, Alarms and Conditions
HA	History Access Types
SNTP	Simple Network Time Protocol
CA	Certificate Authority
CRL	Certificate Revocation List
Y2K	Year 2000
TLS	Transport Layer Security
GCC	GNU Compiler Collection
ROM	Read Only Memory
CTT	Compliance Test Tool
I/O	Input Output
SRAM	Static Random-Access Memory
MCU	Micro Controller Unit
IC	Integrated Circuit
HMAC	Keyed-Hash or Hash-based Message Authentication Code
PKCS	Public Key Cryptography Standards
DES	Data Encryption Standard
PEM	Privacy Enhanced Mail
URI	Uniform Resource Identifiers
AES	Advanced Encryption Standard

Table 1 - Terms and Definitions

Introduction to OPC UA

OPC UA is a communications and data modeling standard for the exchange of information over TCP/IP networks. The data exchange is performed between client and server applications. Client applications are typical consumers of process data such as visualization tools like Human Machine Interfaces (HMI) and Supervisory Control and Data Acquisition (SCADA). Server applications are typically producers of process data and provide an interface to data that is contained within a PLC, IO block, custom controller, sensor or any other embedded system.

As per the specification OPC UA application is operating system independent.

Figure 1 represents typical OPC UA communication between the **OPC UA Server** and **OPC UA Client** applications.

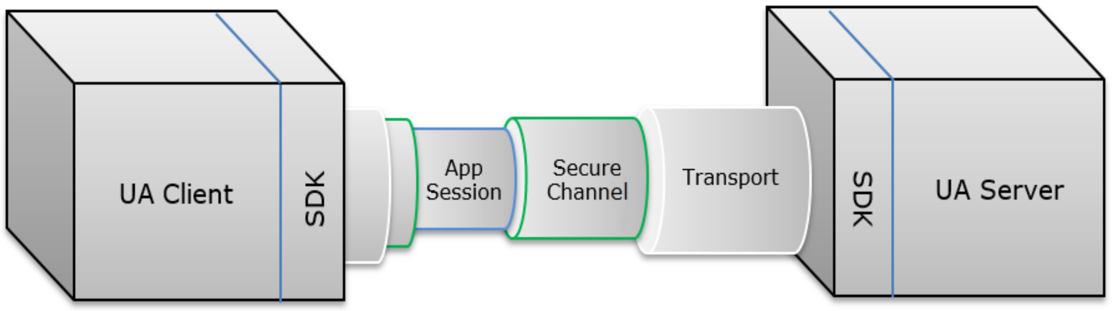


Figure 1 – OPC UA communication

SDK is the Software Development Kit which helps the application developer to develop the OPC UA application within a short duration and with minimal or no knowledge on OPC UA.

Transport is the communication medium which is used to communicate between OPC UA applications.

A *SecureChannel* is a long-running logical connection between a single Client and a single Server. Between a server and a client there should be only one secure channel at any given time.

App session is a specific pathway associated with a specific user where the complete data is transferred. Multiple sessions inside a secure channel is possible.

The OPC UA Specification is found in the OPC Foundation website mentioned below:

https://opcfoundation.org/.

OPC UA Applications

Server or Client Applications is created following the OPC UA standard. Any Application which follows the OPC UA standard is interoperable with any other OPC UA application irrespective of the vendor, Hardware or the platform.

OPC UA Server Application

An OPC UA Server is an application written with the help of an SDK, which exposes data from devices to an OPC UA Client.

Figure 2 (from specification Part 1, **Overview and Concepts**, Section 6.3.1 **Figure 5** - OPC UA Server architecture) represents an OPC UA Server and shows its key components.

Note:

Refer to **Overview and Concepts** document, where the key components are listed.

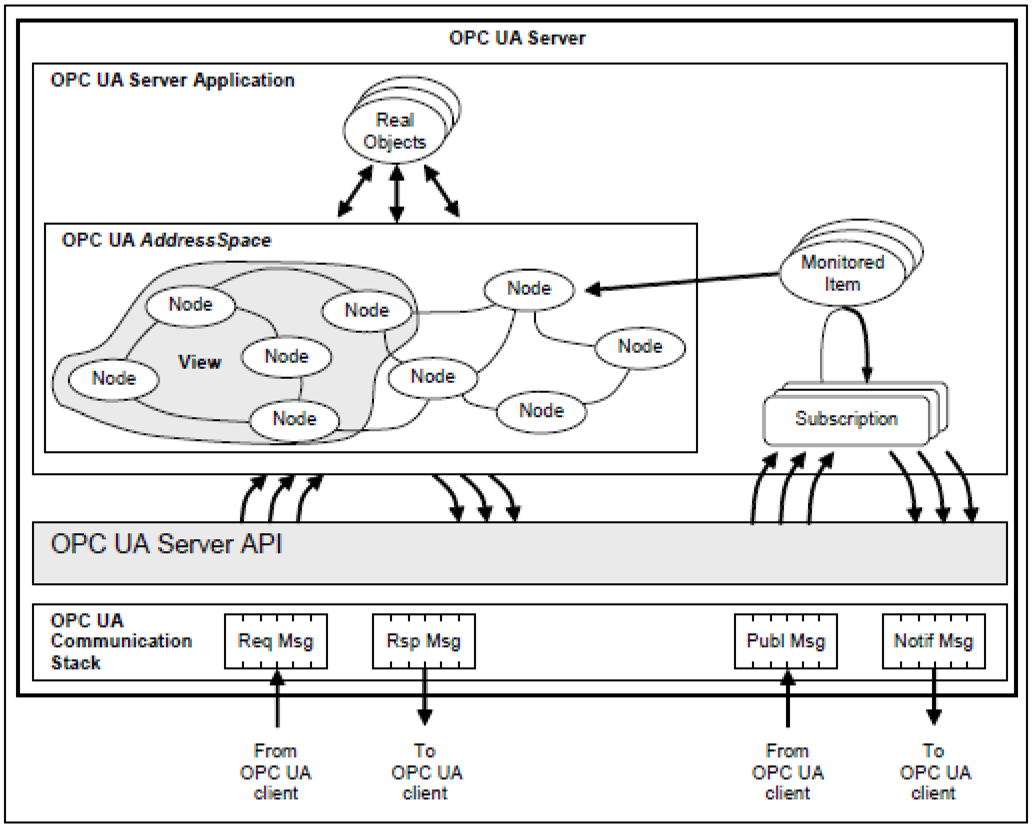


Figure 2 – OPC UA Server

Real objects are processes or data variables that the server exposes. These objects are mapped to nodes in the address space. Each node contains attributes and these attributes contain information pertaining to the node. The most important attribute of a node is the value attribute which represents the value of the real object.

The address space or a subset of the address space called a view is explored by OPC-UA clients. When a client identifies a node attribute that it wants to read or write, it sends read and write request messages to the server. These requests are handled by the OPC-UA communication Stack, a core component of the Device Server SDK software library.

If a client wants to monitor a node attribute for changes in state or value on a periodic basis, the client can create a *subscription* in the server and add *monitored items* to the subscription, each monitoring a specific attribute of a node. The monitored items sample the node attributes and queue the results. The subscription communicates the results to the client on a periodic basis.

Sessions and Requests

When a client successfully connects to an OPC UA Server a *session* is established. This session provides state information related to the communication between the client and the server. The session is managed and maintained internally within the Device Server SDK software library.

From a user's perspective sessions are handled transparently by the library. The communication activity between a client and the application takes place in the context of a session. This activity is always composed of *request messages* from the client and *response messages* from the server.

Subscriptions and Monitored Items

If a client wishes to monitor values in a periodic fashion, a *subscription* with the server is established. The client adds *monitored items* to the subscription, each specifying a variable attribute to be monitored by the server. The server samples each monitored item at a negotiated interval known as the sampling interval.

The client sends *publish* request messages to the server and these requests are queued internally in the subscription. The server periodically receives a publish request and sends a response containing notifications of data changes. This happens at a negotiated interval known as the publishing interval (typically 100ms – 5000ms). The data change notifications contain data values that were recently sampled by the monitored items.

From a user's perspective subscriptions and monitored items are handled transparently by the library. Whenever the library needs to read a data value to perform sampling of a monitored item it executes the read callback to get the latest data value from the host application. There is no distinction between a read callback executed in response to an asynchronous read request from a client and a read callback executed in response to the sampling of a monitored item.

Address Space

The address space is the core of the OPC UA Server. It is a representation or model of the real data that the server exposes.

Nodes, References, Attributes and Properties.

The address space is constructed of *nodes* and *references*. Nodes in the address space contain *attributes* and *properties*.

References are the links that connect nodes together. There is great flexibility in how nodes can reference each other and rather than a strictly hierarchical model, OPC-UA also allows circular references.

Attributes are the information that makes the node useful such as the value of a variable, the read and write permissions of the variable, a textual description of the variable, etc.

Properties are standard nodes that a node references. For example, an *AnalogItem* node has a *EURange property*. The *EURange* property is a child node that contains an upper and lower operating limit for the value exposed by the *AnalogItem* in the Engineering Units of the *AnalogItem. AnalogItem* and all other relevant node types are explained in this document.

Views

Some nodes represent *views*. *Views* are a type of folder that contains references to other nodes. If a client starts navigating the address space from a view, that view can present nodes to the client that are most relevant to the task, the user of the client wants to perform. For example, a server may contain process values that are interesting to an operator and configuration variables that are interesting to a technician. These sets of data can be exposed through different views with different read and write permissions to ensure that the end user is presented with data in a way that is sensible to them.

The set of nodes in different views can be overlapping and means different views represents the same data in different ways or represent different data, or a combination of both.

Folders

Folders are a type of node that contains other nodes. When it says contains other nodes the reference is made to other nodes but can think of them in terms of containing other nodes like a folder contains documents.

Variables

Variables are the most important nodes in the address space as they are the nodes that expose the process data to the client. There are many types of variable nodes and some node types can themselves contain many different types of the data payload. The variable types that are user configurable in the Embedded Device Server Software Stack are described below.

Variable Types

OPC-UA supports numerous variable types. Variable types, refer to the different types of address space nodes that can contain process data values. This is distinct from the data type of the process data value that the variable node contains. For example, a *BaseDataVariable* is a specific node or variable type in the address space. The *BaseDataVariable* node type can, however, contain a variety of data types within its value attributes, such as a 16-bit signed integer or a double precision floating point value.

BaseDataVariable

A *BaseDataVariable* is the simplest way to model process data in an OPC-UA Server. This is a simple node type that exposes a value to the client. The value can be a scalar or an array and a variety of data types including strings and byte arrays.

OPC UA Client Application

An OPC UA Client is an application written with the help of the SDK, which requests data from an OPC UA Server application.

The *Client-Application* is the code that implements the function of the Client. It uses the *OPC UA Client API* of the SDK to send and receive OPC UA Service requests and responses to the OPC UA Server. The *OPC UA Communication Stack* is the internal layer underneath the *OPC UA Client API* of SDK and it converts *OPC UA Client API* calls into Messages and sends them through the underlying communications entity to the server at the request of the client application.

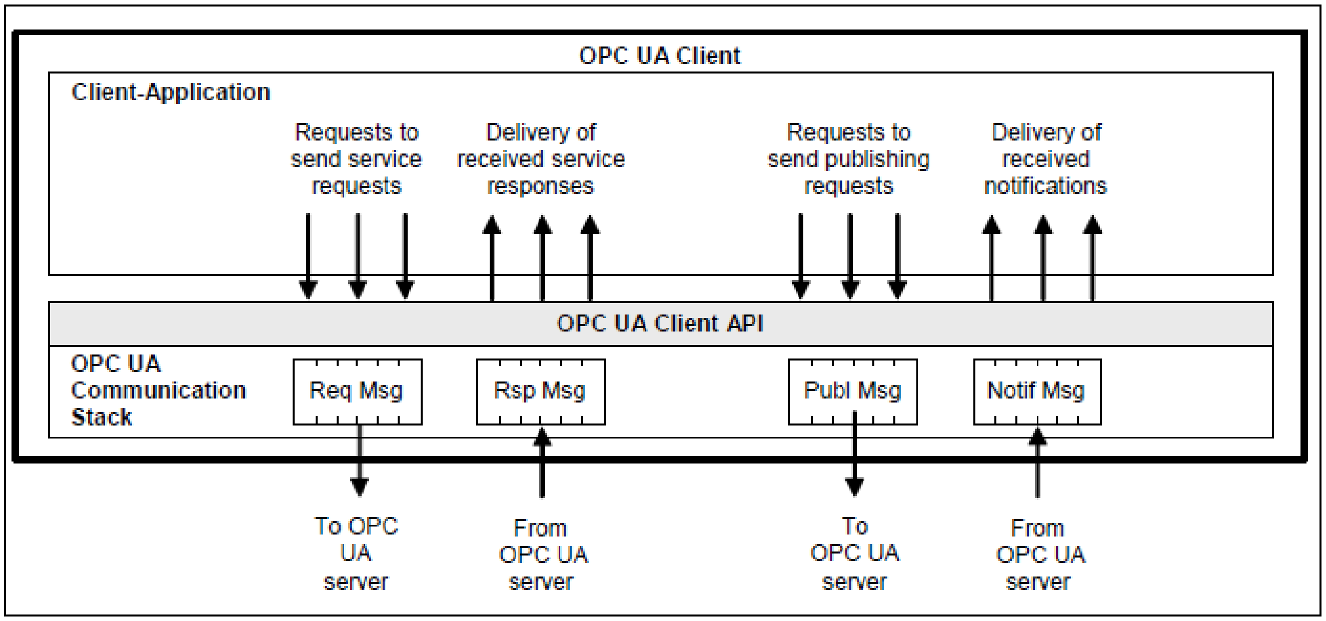

OPC UA Client
Client-Application
Requests to send service requests
Delivery of received service responses
Requests to send publishing requests
Delivery of received notifications
OPC UA Client API
OPC UA Communication Stack
Req Msg
Rsp Msg
Publ Msg
Notif Msg
To OPC UA server
From OPC UA server
To OPC UA server
From OPC UA server

Figure 3 – OPC UA Client

The *OPC UA Communication Stack*, also receives the response and notification messages from the underlying communications entity and delivers them to the client application through the *OPC UA Client API*.

Security in OPC UA

OPC UA provides security by authenticating clients and servers and encrypting communications using X.509v3 application instance certificates. Asymmetric public key algorithms are used for symmetric key generation and exchange periodically, with most client/server communication secured with symmetric algorithms.

A secure server must be provisioned with a certificate during commissioning. This can either be done by generating a default self-signed certificate on the device or by storing a certificate on the device that is generated elsewhere.

Note:

It is strongly recommended that the end user of the OPC UA product install a corporate signed certificate. The use of a self-signed certificate is highly discouraged.

Once a secure channel is established between a client and server, user authentication and permissions are then handled at the session level. User authentication is provided through anonymous users, username/password combinations, and/or user certificates.

Profiles

A profile represents full-featured definition of functionality that must be supported. A Facet is a single piece of defined functionality. Profiles are collections of Facets and other Profiles. The OPC Foundation describes profiles online in this **link**.

Profiles are also defined in the OPC Specification. Refer to the **OPC UA specification Part 7** for profile information.

Introduction to Matrikon® Flex OPC UA SDK

The Matrikon® Flex OPC UA SDK is a cross-platform source code that developers can use to add OPC UA functionality to their applications. **Figure 4** represents the cross-platform OPC UA application.

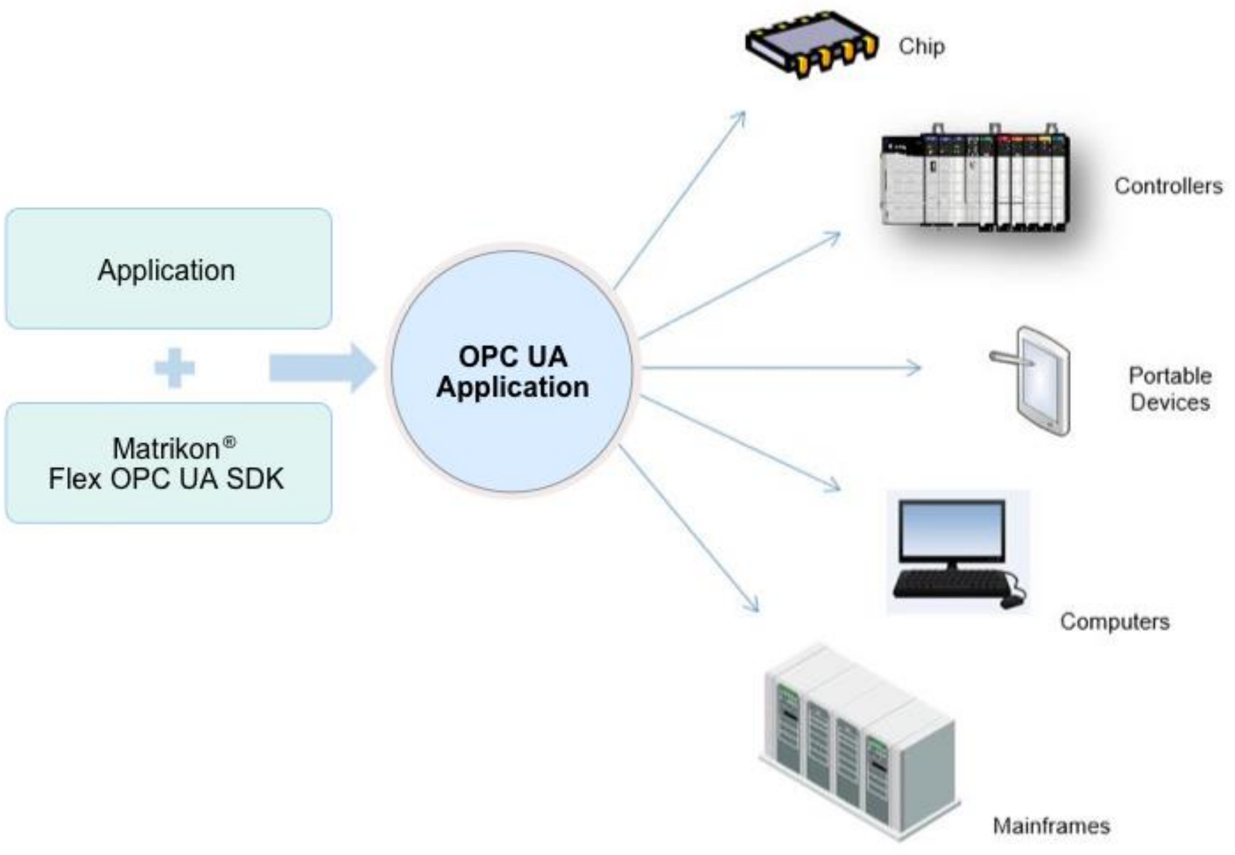


Figure 4 – OPC UA Cross-Platform Application

The Matrikon® Flex OPC UA SDK is a Software Development Kit for OPC UA Servers and/or UA Clients with the following benefits:

- A small code and data footprint suitable for deployment on single-chip microcontrollers or multi-core server class enterprise systems
- Extremely fast and efficient implementation
- User friendly API which facilitates quick integration with the SDK and the device
- Robust and reliable software architecture
- Supports single and multi-threaded configurations
- Customized for use of cryptographic and XML libraries
- Compatible with 32-64 bit

Using Matrikon® Flex OPC UA SDK, Server as well as Client application can be developed. **Figure 5** represents typical OPC UA Server/Client created using SDK.

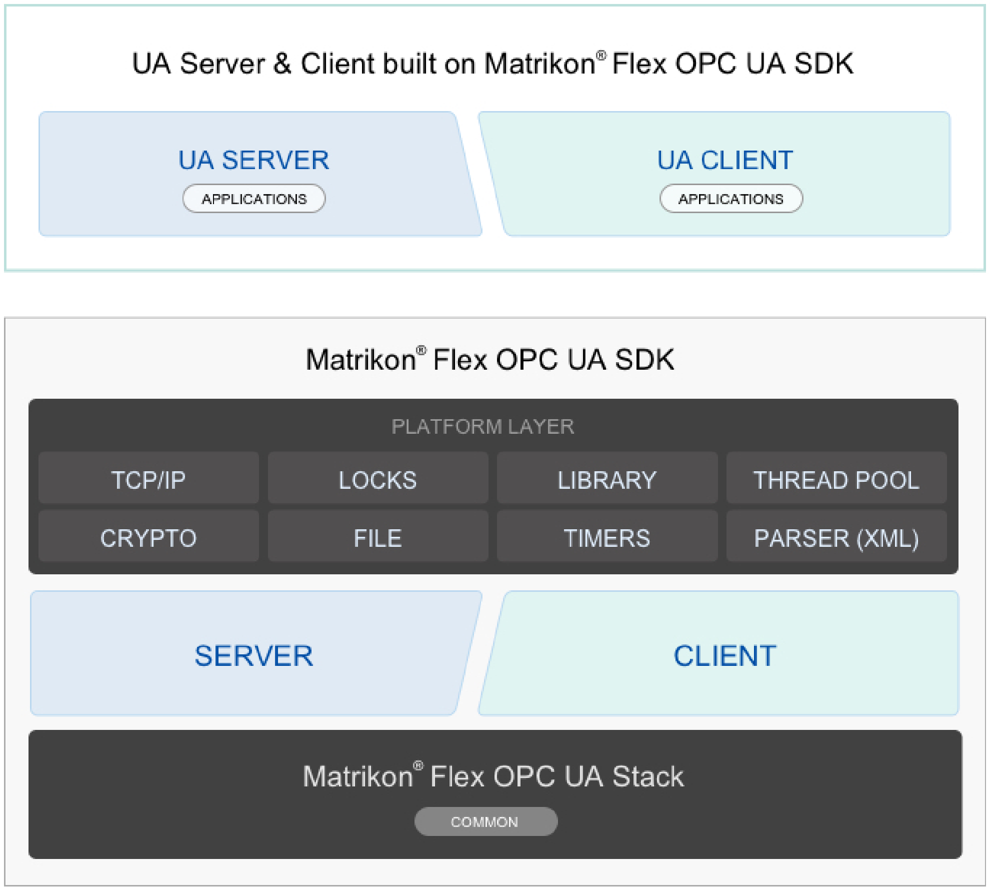


Figure 5 – OPC UA Server and Client

SDK Specification

This section describes the specifications that Matrikon® Flex OPC UA SDK supports.

Supported Profiles

Server

- Standard Server Profile
- Standard Event Subscription Server facet
- Address Space Notifier Server Facet
- A&C Basic Condition Server Facet
- A&C Refresh2 Server Facet
- A&C Address Space Instance Server Facet

- A&C Enable Server Facet
- A&C Alarm Server Facet (some options not supported)
- A&C Acknowledgeable Alarm Server Facet
- A&C CertificateExpiration Server Facet (some options are not supported)
- Auditing Server Facet
- ComplexType Server Facet
- Data Access Server Facet
- History Access

Client

- Core Client Facet
- Base Client Behaviour Facet
- Discovery Client Facet
- Multi Server Client Connection Facet
- Address Space Lookup Client Facet
- Attribute Read / Write Client Facet
- Data Change Subscriber Client Facet
- Data Access Client Facet
- Method Client Facet
- State Machine Client Facet

Supported Transport Protocol

UA-TCP UA-SC Binary (commonly known as opc.tcp)

Supported Security Facet

- SecurityPolicy – None
- SecurityPolicy – Basic128Rsa15
- SecurityPolicy – Basic256
- SecurityPolicy – Basic256Sha256
- SecurityPolicy - AES128SHA256_RSA_OAEP
- SecurityPolicy - AES256SHA256_RSA_PSS

Limitation

There is no limit to the number of concurrent sessions, subscriptions, monitored items and address space nodes (subject to available resources).

Address Space

- Information model creation through:
    - XML Nodeset file import, update and export
    - Programming API
    - ROM/Flash based nodeset import

- Complex Data Type, Complex Objects and Complex Variables support
- Methods support

User Permissions

User permission model is mapped to any backend user authentication system.

Data Source Integration

- Process data is integrated into the SDK through:

    - Direct storage in address space variable nodes
    - Synchronous callback implementation
    - Asynchronous callback implementation to support slow data sources

Platform Requirement

This section describes the platform support requirements for implementing the Matrikon® Flex OPC UA SDK.

CPU

A 32 bit or 64-bit CPU.

Compiler

A C++ compiler that is compatible with the ***ISO/IEC 14882***:1998 (C++98) standard. The code base uses C99 fixed with datatypes. If these are not defined by the toolchain they must be defined for the user's platform.

Note:

SDK is normally implemented in C++, if required the application can also be implemented in C. An optional C API is provided for easy usage of basic server features from a C application. But to optimize the complete SDK functionality, it is recommended to use C++ API.

IEEE 754 Floating Point Conformance

The SDK requires a runtime environment that conforms to IEEE 754.

Note:

Certain compiler options such as GCC "fastmath" can impair this conformance resulting in undefined behavior. If the system does not provide a conformance guarantee, careful testing is required to confirm expected server behavior with floating point quantities.

TCP/IP Stack

The SDK API has been designed to facilitate integration with any TCP/IP stack. On high-performance platforms, TCP/IP stacks typically provide a socket API and example implementations are provided in the SDK Distribution for integration with both Berkley and Windows sockets.

Many embedded TCP/IP stacks do not provide sockets but provide a block-based API instead. The SDK can be integrated with any block-based API.

For microcontrollers examples integrating with the open source TCP/IP stack LwIP (**http://savannah.nongnu.org/projects/lwip/**) in both raw and socket mode are provided. It is recommended, however, that customers use a commercial TCP/IP stack in their product.

When choosing an embedded TCP/IP stack, care must be taken to ensure that the stack is full featured. Specifically, where OPC UA subscriptions are required then the TCP/IP stack must support multiple unacknowledged TCP segments in flight. Some embedded TCP/IP stacks such as the uIP TCP/IP stack do not support this behavior and are not suitable as a basis for a compliant OPC UA Server that supports subscriptions.

The TCP/IP stack must also support multicast to facilitate support of the OPC UA Local Discovery functionality which shall be released as an extension to the standard shortly.

Time Synchronization

All OPC UA Clients and Servers require the host system to provide them with date and time information for time stamping of various server activities. Time is depicted in the following methods:

- Battery Backed Real Time Clock
- Simple Network Time Protocol (SNTP) - A SNTP Client application is integrated into the device. Such clients can either listen to SNTP broadcasts from a time server located on the LAN or alternatively request time updates periodically from specific SNTP servers. Several free NTP server applications are available for download and it is a normal practice in Industrial Automation networks to have an NTP Server present to synchronize time amongst PC based OPC Clients and Servers. Sample source code is available demonstrating a reference implementation of an SNTP Client written in ANSI C. An SNTP Client that listens to time broadcasts can be implemented in a few simple lines of code.
- IEEE15888 PTP – Instances where high-resolution time synchronization is required, IEEE1588 Precision Time Protocol is used if the network infrastructure supports it and support has been designed into the device. Most microcontrollers that integrate an Ethernet MAC, provide hardware support for this feature that can provide time synchronization to the order of 100ns between devices on a LAN or even WAN (through GPS clock synchronization).

Runtime Environment

The SDK can be built in both single-threaded and multi-threaded configurations. A single-threaded build can run bare metal in a loop in main () on a microcontroller or in a single-thread or task of an OS. Such configuration is not thread-safe and calls to the SDK must be made on the same thread.

A multi-threaded configuration creates its own thread during initialization. Such a configuration is completely thread-safe and the application developer can call the SDK from multiple threads if certain rules are implemented.

Security in SDK

Refer to the **Security in OPC UA** section to understand the Concepts of Security in OPC UA.

Refer to the **Security Related Configurations** for configuring the OPC UA Application with security.

Secured Communication

Note:

The server closes the secure channel/session involved with the client, even during security threat attacks by a trusted client. It does not have any effect on other existing channel(s)/session(s).

Security Considerations

A cohort document titled **Matrikon® Flex OPC UA SDK Security Considerations – R410.3.pdf** contains important information about maintaining secure practices for developing and deploying the product. It is recommended to read and understand this document before developing any product with the Flex SDK.

Note:

Flex SDK provides sample projects and code. Although the samples have been tested they have been written for clarity of the concepts expressed in the example and has not been implemented following all security practices. The application developer is recommended to test the code using the user’s corporate secure coding practices and procedures.

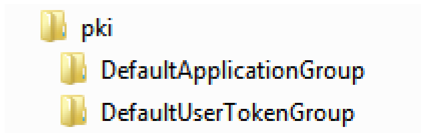
Integrating Security

The SDK provides transport layer security by integrating an open cryptographic API with a third-party cryptography library. OpenSSL integration is supported on Linux and Windows platforms, and multi-platform support is available through MBedTLS and NanoSSL integration. Set the relevant macros in the configuration header to 1 and link to the chosen cryptography library to integrate security. Linux and Windows examples are provided as part of the SDK distribution.

Refer to **this** section for APIs related to security.

Certificate Stores

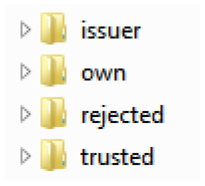
The certificate store (pki) folder structure is as follows:



There are two certificate groups "*DefaultApplicationGroup*" and "*DefaultUserTokenGroup*" in the folder structure as depicted above.

- **DefaultApplicationGroup**: The certificate(s) which is/are used for endpoint Sign, Sign and Encrypt security policy are stored here.
- **DefaultUserTokenGroup**: The certificate(s) which is/are used for providing User permission (x509 v3 certificate User Token Policy) are stored here.

The pki folder contains other groups along with the two groups mentioned above. Every group contains the same folder structure shown below.



- **Issuer folder** contains the Certificates and CRLs from Certification Authority or Issuer.
- **Own folder** contains the Self-signed certificate and private key. This folder must be protected with file permissions so that the private key cannot be obtained by an attacker.
- **Rejected folder** contains the rejected Certificates.
- **Trusted folder** contains the trusted Certificates and CRLs.

Note:

Private key creation will not be successful if the private key is not password protected. Minimum length of the password for private key is 1.

Endpoint Configuration

A server endpoint is a logical point where a client attempts to connect. A single server can have multiple endpoints implementing different security policies. Each endpoint accepts several different user token types (anonymous, username/password, etc.). Each endpoint type enables the configuration by adding one or more user token policies to the endpoint. This is carried out via a bitmap of user token policies. Setting the user token policy bitmap to zero for an endpoint completely disables it.

The security configuration data structure contains a field for each supported endpoint security policy. An endpoint security policy is a predefined communication mode that mandates a combination of security algorithms and optionally message signing and encryption.

The supported endpoint security policies are as follows:

- Endpoint Security Policy - None
- Endpoint Security Policy - Sign - Basic128Rsa15
- Endpoint Security Policy - Sign & Encrypt - Basic128Rsa15
- Endpoint Security Policy - Sign - Basic256
- Endpoint Security Policy - Sign & Encrypt - Basic256
- Endpoint Security Policy - Sign - Basic256sha256
- Endpoint Security Policy - Sign & Encrypt - Basic256sha256
- Endpoint Security Policy - Sign - AES128SHA256_RSA_OAEP
- Endpoint Security Policy - Sign & Encrypt - AES128SHA256_RSA_OAEP
- Endpoint Security Policy - Sign - AES256SHA256_RSA_PSS
- Endpoint Security Policy - Sign & Encrypt - AES256SHA256_RSA_PSS

Note:

It is not recommended to enable **Endpoint Security – None** as it provides no protection from tampering or snooping of OPC Communication.

User Permissions

The application developer must implement the **IUserPermissionsProvider_t interface** and register it with the server. The server will then call the permissions provider each time a user attempts to activate a session on the server. The provider authenticates the user credentials and if the user is authorized, the server returns an implementation of the **IUserPermissions_t interface** to the SDK. The SDK uses this user permissions object to authenticate all attempts to access nodes in the server address space.

The user authentication token types currently supported by the SDK are:

- Anonymous
- Username/Password
- X509 v3 Certificates

Each field is a bit map of user token policies. User token policies are a combination of security algorithm and user credentials that allow a user to login to the server and activate a session.

Enter zero in the field value to disable an endpoint security policy completely. If the user wants to enable a security policy, add a user token policy to the security policy by specifying a bitmap of enabled user token policies.

The supported user token policies are:

- User Token Anonymous - Token Security Policy None

- User Token Username / Password - Token Security Policy None
- User Token Username / Password - Token Security Policy Basic128Rsa15
- User Token Username / Password - Token Security Policy Basic256
- User Token Username / Password - Token Security Policy Basic256Rsa256
- User Token Username / Password - Token Security Policy AES128SHA256_RSA_OAEP
- User Token Username / Password - Token Security Policy AES256SHA256_RSA_PSS
- User Token X509v3 Certificate - Token Security Policy None
- User Token X509v3 Certificate - Token Security Policy Basic128Rsa15
- User Token X509v3 Certificate - Token Security Policy Basic256
- User Token X509v3 Certificate - Token Security Policy Basic256Rsa256
- User Token X509v3 Certificate - Token Security Policy AES128SHA256_RSA_OAEP
- User Token X509v3 Certificate - Token Security Policy AES256SHA256_RSA_PSS

If the user is using **user token username / password** for unencrypted endpoint security policies, it is recommended to enable user token policies that only encrypt the password. For other endpoint security policies, there is no need to encrypt the user password as the secure channel is encrypted.

In the case of encrypted endpoint security policies, enable only **username/password - token security policy none** if the entire address space is intended to be open to all users who can validly connect to the UA Server.

If using **X509v3 Certificate** user token policies, it is recommended to use only token security policies that require signing (see Part 4 of the OPC UA specification or contact support for more information).

Note that all X509 certificates must be stored in the certificate stores in DER format only.

Address Space

This section describes the OPC UA Address space and its objects. The primary objective of the OPC UA Address space is to provide a standard way for the servers to represent objects to the clients. The OPC UA object model has been designed to meet this objective. It defines objects in terms of variables and methods. It allows connectivity to other objects to be expressed as well.

The set of objects and related information that the OPC UA Server makes available to clients is referred to as its Address space. Objects and their components are represented in the Address space as a set of nodes described by attributes and are interconnected by references.

SDK Address Space Design

The SDK address space design are separated into two distinct and complementary parts, the low-level address API, and the helper classes. The address space API is designed to be a reliable reproduction of Part 3 of the UA specification and contains the bare minimum functionality. This is to minimize the amount of work an application developer must do to implement their own address space in the future. Most application developers will not implement an address space but will use the default implementation provided with the SDK. There are some cases, however, where the user may have to implement their own address space. For example, an enterprise server wants to store the address space metadata in a SQL database rather than internally.

Since the address space API is low level, it can take several steps to create a complex variable and configure all its attributes and references. To make this easier, helper classes have been created for common complex variables such as AnalogItems from the Part 8 Data Access specification. The helper classes can be used to create complex variables in the address space and to wrap pre-existing complex variables in the address space to provide a higher level of abstraction which in turn increases their ease of use.

Application developers create their own helper classes to make interaction with their proprietary complex variables easier.

The default address space is completely thread-safe when the SDK is configured for multi-threaded operation and can be accessed asynchronously by the application developer from any thread at any time.

Configuring SDK (Build Configuration)

Some of the modules like TCP/IP, Threads, Lock, etc., need to have different implementations for different platforms for optimization and efficiency based on the support provided by the platform. In SDK, there are implementations available for legacy C API, Single-Threaded and Multi-Threaded environment for Windows and Linux platforms. These implementations can be referred for efficient and optimized implementation of modules for any other platform.

UASDK Platform Layer Components

The platform layer components that must be implemented are as follows:

1. **Asynchronous Operation**: Operations that need to be completed asynchronously can use this implementation
2. **Atomic variables**: Variables that can be accessed across multiple threads safely.
3. Crypto Library: Cryptographic support is provided by optional cryptographic libraries. Support is provided out of the box for OpenSSL, mbedTLS, and NanoSSL. Other libraries can also be used; however, the binding must be implemented by the application developer. It is

easy to modify the provided bindings to support hardware acceleration, secure key storage, etc., where the platform supports it.

4. **Directory Abstraction**: For handling the Certificates for certificate management, requires a directory abstraction.
5. **File Abstraction**: Using the XML IO functionality requires a file abstraction.
6. **Library Calls**: A variety of library calls are required to be implemented. Access to the C standard library functionality, which provides platform time support for the SDK's internal timing and time stamping requirements, is required.
7. **Locks**: Locks are required for locking the access to shared resources in a multi-threaded configuration.
8. **Reference Counts**: The SDK uses boost-style intrusive reference counting smart pointers extensively. The reference counts must be implemented to be thread-safe on a given platform.
9. **TCP/IP**: The Platform TCP/IP connection abstraction must be implemented. Check the section **TCP/IP Stack**.
10. Thread
    Creating and handling the threads
11. **Thread Pool**: The thread pool provides support for executing workloads across a thread or threads.
12. **Timers**: Timers provide timing functionality required internally within the SDK.
13. **XML**: Nodeset import through XML is provided via TinyXML2 or LibXML2 out of the box or alternatively, another library.

The TCP/IP interface is the most complex part of the API. The complexity is the result of two requirements. Firstly, having to enable the use of block-based as well as socket based TCP/IP stacks and, secondly, to minimize the copying of data to maximize performance on resource-constrained hardware.

Sample code is provided to map the API to Windows sockets, Berkeley sockets and to the LwIP raw API. If the available TCP/IP stack API is significantly different from these samples, the user developer must perform the integration themselves.

Single versus Multi-Threaded

The effort required to integrate the SDK to a new platform varies depending on whether it is a single or multi-threaded configuration. The different types of configurations available are:

- Single-Threaded Environment
- Multi-Threaded Environment
- Backward Compatibility (C Api)

Note:

Refer the following links to know how to configure **Single-Threaded Environment**, **Multi-Threaded Environment** and **Backward Compatibility (C Api)**.

Advantages and Disadvantages

Advantages of single-threaded configuration are:

- Easy to port as very few platform specific primitives need to be implemented.
- The single-threaded platform layer provided will run on virtually any platform without modification.
- Simple and reliable.
- Lack of thread synchronization yields very high performance on slower embedded systems.
- Easy programming model. Performs all interactions with the SDK in a single-thread and potentially avoids any concern about race conditions or locking requirements at the application level. Less chance of bugs in the application code makes development faster.

Disadvantages of the single-threaded configuration are:

- Server monitored item sampling can exhibit a high degree of jitter due to service calls being made by clients being handled at the time when sampling is required (this is typically not as much of an issue as may appear as the actual data source sampling and time stamping is performed independently by the application and can be hard real time).
- When using security, the channel and session handshaking can take hundreds of milliseconds on a slow microcontroller and while this is happening no other activities can take place. This can make the level of jitter unacceptable depending on application requirements, although, this is not common.
- When using security on a slow CPU and supporting multiple concurrent sessions a channel/session handshake on a new session will stall subscriptions on an existing open session and this may result in a compliance failure if the product requires OPCF compliance approval certification

Advantages of the multi-threaded configuration are:

- When a server is deployed on a multi-core system, the server can be easily scaled to use many cores in parallel to harness the full power of the platform.
- The OS will schedule the workload so that Clients see excellent interactive responses. For example, even on a single core system, a long channel/session handshake will be pre-empted by the OS to perform other activities such as sampling, handling requests from other Clients, etc.
- It facilitates easy interaction with the server and the address space across multiple threads.

Disadvantages of the multi-threaded configuration are:

- More complex programming model as callbacks is called on worker threads requiring the user to synchronize access to their data sources and any other shared data in their application.
- More difficult to port as a variety of platform specific primitives must be implemented.

Thread synchronization overhead slows down the server on single core platforms.

Configuring Single-Threaded Environment

Set the below macros for using the modules in Single-threaded environment. Before configuring, refer to the **Single versus Multi-Threaded** section for more information.

- **UASDK_INCLUDE_ASYNC_OPERATION_ANY_PLATFORM_ST**

- **UASDK_INCLUDE_ATOMIC_ANY_PLATFORM_ST**
- **UASDK_INCLUDE_DIRECTORY_WINDOWS**
- **UASDK_INCLUDE_FILE_WINDOWS_LINUX**
- **UASDK_INCLUDE_LIBRARY_WINDOWS_LINUX**
- **UASDK_INCLUDE_LOCKS_ANY_PLATFORM_ST**
- **UASDK_INCLUDE_REF_COUNT_ANY_PLATFORM_ST**
- **UASDK_INCLUDE_NETWORKING_WINDOWS_LINUX_SINGLE_THREADED**
- **UASDK_INCLUDE_THREAD_POOL_ANY_PLATFORM_ST**
- **UASDK_INCLUDE_TIMERS_ANY_PLATFORM_POLLED**

Configuring Multi-Threaded Environment

Set the below macros for using the modules in Multi-Threaded environment. Before configuring, refer to the **Single versus Multi-Threaded** section for more information.

Note: In Windows environment to get the full benefit of Windows, developer can configure application like mentioned below. Windows Developer can also use Multi-Threaded or Single-Threaded based on the requirement.

- **UASDK_INCLUDE_ASYNC_OPERATION_CPP11 / UASDK_INCLUDE_ASYNC_OPERATION_WINDOWS**
- **UASDK_INCLUDE_ATOMIC_CPP11 / UASDK_INCLUDE_ATOMIC_WINDOWS**
- **UASDK_INCLUDE_DIRECTORY_WINDOWS**
- **UASDK_INCLUDE_FILE_WINDOWS_LINUX**
- **UASDK_INCLUDE_LIBRARY_WINDOWS_LINUX**
- **UASDK_INCLUDE_LOCKS_CPP17 / UASDK_INCLUDE_LOCKS_CPP11 / UASDK_INCLUDE_LOCKS_WINDOWS**
- **UASDK_INCLUDE_REF_COUNT_CPP11 / UASDK_INCLUDE_REF_COUNT_WINDOWS**
- **UASDK_INCLUDE_NETWORKING_WINDOWS_LINUX_MULTI_THREADED**
- **UASDK_INCLUDE_THREAD_CPP11 / UASDK_INCLUDE_THREAD_WINDOWS**
- **UASDK_INCLUDE_THREAD_POOL_CPP11 / UASDK_INCLUDE_THREAD_POOL_WINDOWS**
- **UASDK_INCLUDE_TIMERS_CPP11 / UASDK_INCLUDE_TIMERS_WINDOWS**

Configuring Backward Compatibility (C API)

Set the below macros for using the modules in Legacy C API based server application.

- **UASDK_INCLUDE_ASYNC_OPERATION_ANY_PLATFORM_ST**
- **UASDK_INCLUDE_ATOMIC_ANY_PLATFORM_ST**
- **UASDK_INCLUDE_LIBRARY_C_API**
- **UASDK_INCLUDE_LOCKS_ANY_PLATFORM_ST**
- **UASDK_INCLUDE_REF_COUNT_ANY_PLATFORM_ST**
- **UASDK_INCLUDE_TCPIP_C_API**
- **UASDK_INCLUDE_THREAD_POOL_ANY_PLATFORM_ST**
- **UASDK_INCLUDE_TIMERS_ANY_PLATFORM_POLLED**

Security Related Configurations

OPC UA describes a secured way of communication between server and client application. How to achieve secured communication, and how to integrate the different security bindings with Matrikon® Flex OPC UA SDK is described below.

User Authentication

User authentication can be provided through anonymous users, username/password combinations, and/or user certificates. The server must implement the APIs provided by the module **IUserPermissionsProvider_t** to verify the user token provided by the client. Refer to the **User Authentication APIs** section for **IUserPermissionsProvider_t** APIs.

Initial Provisioning

Warning:

Each Server and/or Client must possess a unique application instance certificate. The device vendor must decide how to fulfill this requirement. There are three options:

- SDK creates a default, self-signed certificate and associated key pair when it first starts up and/or when the device network configuration (IP Address or Hostname) is changed.

  The advantage of this approach is that it requires little intervention by the end user and the certificate private key is not exposed to any third-party system where it could be compromised. The disadvantages are that it requires very effective entropy generation on the device which is not always possible. Application instance certificates are long lasting and the strength of the security they provide is only as strong as the strength of the key pair used to sign them. In addition, the certificate generation can take a long time and this will influence initial device boot time.

- SDK creates a default, self-signed certificate on the production line and installs it on the device.

  The advantage of this approach is that it requires little intervention by the end user and does not require high-quality entropy on the device. The disadvantage is that the certificate private key is exposed to a third-party system (in production) where it could be compromised.

- SDK ships the device with no certificate and the end user needs to provision the device with either a self-signed or CA issued certificate during commissioning.

  The advantage of this approach is that no certificate generation is required in production or on the device. The disadvantage is that the end user cannot connect to the device securely without provisioning the device with a certificate.

Even in the case of 1 or 2 above, the end user may still wish to replace the default certificate to use CA issued certificates which may suit his use case better.

Certificate Validation Options

The security API applies options while validating certificates that are configured as “certificate validation option bits” as defined in the OPC UA specification. The validation option bits are configured by the application developer by configuring a **CertificatevalidationOptions_t** object and placing it into the server configuration. The SDK then translates this into option bits for the security components.

Note that the online validation of CRLs (Certificate Revocation Lists) is not currently supported.

Trusting All Certificates at Server Side

To facilitate initial provisioning, the application can set the **trustCertificateIfNotInTrustList** value to **True** in *ICertificateStoreCallback_t::RemoteCertificateAboutToBeValidated()* API In C++.

For CAPI **UASecurity_Set_validation_options2()** function needs to be called with the required value. This allows the first client to connect to a server to configure the server when the trust list is empty.

When connecting in this way, an encrypted connection is created but there is no authentication (any client can connect). The authentication for the connection must be handled at the session level where the configuration client must provide user credentials such as a unique or one-time administrator password to complete the connection.

Self-Signed versus CA Issued Certificates

Certificates are created by Certificate Authority (CA) or within the application (Self-Signed). Advantages and/or Disadvantages of the same are described below.

Self-Signed

- Certificate Revocation Lists (CRL)s are not required
- Device trust list must be updated for every new application that requires a connection
- Device trust list can become cumbersome over time, increasing Flash and RAM requirements

CA Issued

- Only a single CA certificate is required to be installed in the trust list
- Any number of applications can now connect to the device with no additional configuration
- Bounded Flash and RAM requirements
- CRLs become important and the device CRL(s) should be updated regularly

Security Error Suppression

Certain certificate errors are suppressed by the application using the API. Specifically, the application suppresses certificate validity errors and certificate revocation errors. Client applications can also suppress server hostname validation.

Based on a risk assessment, the device developer or end user must set management policies as follows:

- Handling the expired certificates for application as well as for clients
- Handling hostname validation
- Handling the revocation status

⚠ **Warning**:

Setting incorrect policies can have negative consequences such as security breach due to a revoked client certificate and no CRL lookup on a server device, or a Y2K type problem caused by a forgotten device with an expiring certificate.

Saving Rejected Client Certificates

By default, the SDK saves rejected Client certificates in the file system when they pass validation but are not trusted. This provides an easy way to trust new clients by simply moving the certificate to "trusted/certs" folder in the file system.

On very small systems this has the potential to exhaust the file system storage over time and so this functionality can be disabled by configuring the directory actor.

OpenSSL Integration

OpenSSL is typically preinstalled on Linux systems. If not, it can be easily obtained using the distro package manager.

⚠ **Warning:**

If OpenSSL binding is selected for Security library, make sure to use 1.1.1 series or upgrade to 1.1.1 at the earliest. Support for OpenSSL version 1.0.2 and 1.1.0 will end by end of 2019 as per the OpenSSL website (more information can be found in **https://www.openssl.org/source/**).

SDK will continue to keep the 1.0.2 and 1.1.0 changes in our security binding. No fixes for versions 1.0.2 and 1.1.0 will be available after the official support ended.

Entropy Generation

The OpenSSL interface module provided with the SDK attempts to obtain entropy during initialization by reading from "/dev/random". On some embedded Linux systems, this call may block indefinitely due to a lack of available entropy. In some cases, it may work on the device with a certain hardware configuration but when the user makes any changes, it may stop working. Examples of changes could be the removal of a keyboard or network interface.

It is the responsibility of the device vendor and application developer to decide what is the most appropriate entropy source on their system.

Warning:

Operating a server with poor entropy will directly affect the level of security obtained. Both RSA (certificate) and secure channel symmetric key generation require appropriate levels of entropy to provide the expected level of security.

MbedTLS Integration

MbedTLS with SDK requires the the following criteria for integration. On a Linux platform, mbedTLS requires to download the latest library and include this library to the project. The two mandatory folders are:

- mbedTLS > include
- mbedTLS > library

Note:

Other folders can be deleted from the path.

Entropy Generation

In Linux platform, the user can obtain entropy from “/dev/random”. In embedded platforms, a suitable entropy source must be mapped in entropy.c and entropy_poll.c files. On some embedded Linux systems, this call may be blocked indefinitely due to a lack of available entropy. In some cases, it may work on some devices with a certain hardware configuration but when the user makes any changes it may stop working. Examples of such changes could be the removal of a keyboard or network interface.

It is the responsibility of the device vendor and application developer to decide what is the most appropriate entropy source on their system.

Warning:

Operating a server with poor entropy will directly affect the level of security obtained. Both RSA (certificate) and secure channel symmetric key generation require appropriate levels of entropy to provide the expected level of security.

NanoSSL Integration

The integration of NanoSSL with SDK requires the following settings:

Mocana Library Settings:

The settings for the mocana library are as follows:

Required Folder:

- asn1
- common (Exclude all "math_**.s" files from build)
- crypto
- harness
- platform (Exclude "android_rtos.c" from build)

Moptions_custom.h Settings:

Set the moptions_custom.h options. The available options are as follows:

```
#ifndef _RTOS_LINUX_

#define_RTOS_LINUX_

#endif

#define_ENABLE_ALL_DEBUGGING_

#define_ENABLE_MOCANA_PKCS1_

#define_ENABLE_MOCANA_PEM_CONVERSION_

#define_ENABLE_MOCANA_OPENSSL_PUBKEY_COMPATIBILITY_

#define_ENABLE_MOCANA_PKCS5_

#define_ENABLE_MOCANA_SCEPCC_SERVER_

#define_ENABLE_MOCANA_CERTIFICATE_SEARCH_SUPPORT_

#define_ENABLE_MOCANA_LDAP_CLIENT_

#define_ENABLE_MOCANA_URI_

#define_ENABLE_MOCANA_PKCS10_

#define_ENABLE_MOCANA_RAND_ENTROPY_THREADS_

#define_ENABLE_MOCANA_DER_CONVERSION_

#define_ MOCANA_PARENT_CERT_CHECK_OPTIONS_       (0xFFFF)

#define_ MOCANA_SELFSIGNED_CERT_CHECK_OPTIONS_       (0xFFFF)
```

Updation of mrtos.h:

Example for updating mrtos.h with malloc and free callback are as follows:

```
#include< uasdk_c_declarations.h >

#define_ MALLOC UASDK_Malloc

#define_ FREE UASDK_Free
```

Eclipse Project Settings:

Eclipse project settings for mocana library are as follows:

GCC C Compiler:

- Dialect > language- > keep it blank

Symbols > Defined symbols (-D) __LINUX_RTOS__ and POSIX

Warnings > checked only "Pedantic (-pedantic)" and "All warnings"

Includes > Include paths (-I) like i.e., "${workspace_loc:/embedded_profile_with_nanossl/Libraries/mocana/src}"

Generic Build Macros

Matrikon® Flex OPC UA SDK can be configured as per the user requirement. Files to be referred and how to configure SDK has been described below.

Configuration Headers

There are two configuration header files:

- uasdk_default_build_config.h
- uasdk_custom_build_config.h

"uasdk_default_build_config.h" contains all build macros that can be configured for building the SDK and sets as default if they are not already defined. These macros can be defined in the IDE or directly in another header. A placeholder header "uasdk_custom_build_config.h" is provided for this purpose. The placeholder provided in the SDK is empty and is in its own folder. To use this header file, copy it to another folder and add that folder to the user's project and include the path. This approach can be seen in various examples provided in the SDK distribution.

All build macros are prefixed with UASDK_ except for the V2 legacy C API macros which begin with UA_. All macros that represent a Boolean choice must be set to zero or non-zero as defined in the header file comments.

Refer section **Generic Build Macros** in appendix for building macros and its description.

Miscellaneous

Some of the modules which facilitate advanced application creation are described below.

Locales and Translations

The SDK supports an unlimited number of locales. A locale is a language and optionally a region denoted by an identifier string. Example identifiers strings include “en” for English and “en-US” for English (USA).

The SDK defines an ILocalizableText_t interface that can be implemented by the application developer to provide a dynamic translation of any LocalizedText_t file stored in the server address space. A simple implementation of this (LocalizableTextStored_t) is provided which stores the translations internally for each string. If preferred, the application developer can implement this interface to access a ROM-based look-up table of translations or any another suitable backend to perform the translations.

The LocalizedTextStored_t class behaves as described below. The first entry in the object is considered as a default text.

The localized text information can be retrieved from LocalizedTextStored object with the help of APIs in the appendix section **Locales and Translations**.

If status is only the argument, then default text is returned.

If Array of locales is also provided as the input argument, then the first matching locale text is returned.

Both the APIs returns an intrusive reference counting boost-style smart pointer to the localized text. The pointer needs to be tested before dereferencing.

Memory Management

The SDK can be configured to use the system heap for memory allocation. On a PC or enterprise class platform, this should always be preferred as it yields the highest performance and requires no configuration.

Alternatively, on a 32-bit embedded platform, the SDK can be configured to use an internal allocator. The allocator provided with the SDK is optimized for reliability and zero external fragmentation rather than efficiency. When using this allocator, external memory fragmentation is prevented under steady state conditions. Another benefit of the allocator is that all SDK memory allocations are sandboxed from the platform so if the server runs out of memory, the platform itself remains unaffected.

The allocator must be configured and the memory provided must be sized. During development, appropriate SDK limits must be set and maximum memory usage test must be carried out to prevent out of memory conditions occurring in the field.

Refer to the following table for Memory Management guidelines:

	System Heap	**SDK Internal Allocator**
Where to use	On any platform that provides a heap allocator and has lots of extra memory (i.e. desktop, server class machines and embedded systems with an MMU and plenty of RAM).	On embedded platforms with tight memory requirements (i.e. bare metal or RTOS based embedded platforms)
Advantages	• Dynamic memory use • No extra configuration required • Performance	• Fixed maximum memory use determined at compile time. • Prevents allocation fragmentation. • Places all SDK allocations in an effective sandbox. Avoiding any interaction with OS or any applications
Disadvantages	• Can consume system resources. • Reduced performance on embedded systems. • Heap exhaustion can affect more than just the OPC UA application. • External memory fragmentation can occur over time when running with a small heap.	• The required amount of memory must be identified during development. • The entire memory configured for the stack is unavailable for other parts of the application unless the application is written to use the SDK's allocator. • Performance can be slower than optimized system heaps.

Table 2 – Memory Management Guidelines

Logging and Debug

There are many other debug macros available that support logging of different functionality within the SDK. These macros begin with UASDK_DEBUG and are listed below. An implementation of **ILogListener_t** must be completed, instantiated, and statically registered with the Logger_t class. Examples on how to implement this is provided in the SDK distribution.

The logging examples send the output to “stdout” but this can be modified by the application developer (for example, streaming the output to a syslog server).

For Logging enable the build macros described in the Appendix section **Logging and Debug**.

Services specific

Below are some of the macros which are required for specific services.

UASDK_INCLUDE_METHOD_SERVICE_SET

This macro is used to include types for method service set. Exclude to reduce code size.

UASDK_INCLUDE_NODE_MANAGEMENT_SERVICE_SET

This macro is used to include types for node management service set. Exclude to reduce code size.

UASDK_INCLUDE_TRANSFER_SUBSCRIPTIONS_SERVICE

This macro is used to include types for transfer subscriptions service. Exclude to reduce code size.

UASDK_INCLUDE_QUERY_SERVICE_SET

This macro is used to include types for query service set. Exclude to reduce code size.

UASDK_INCLUDE_ADDITIONAL_DISCOVERY_SERVICES

This macro is used to include types for additional discovery services. Exclude to reduce code size.

UASDK_INCLUDE_ADDITIONAL_DA_TYPES

This macro is used to include additional DA types. Exclude to reduce code size.

UASDK_INCLUDE_METHOD_SERVICE_SET

This macro must include methods when subscriptions are selected.

Data Access, Events and History

Below are some of the macros which are required for Data Access, Events and History

UASDK_USE_HELPERS

This macro is used to enable the helper modules for DA or file object.

UASDK_USE_DATA_ACCESS_HELPERS

This macro is used to include the DA related nodes. Set the value to 0 to exclude the data access related nodes and 1 to include the data access related nodes.

UASDK_INCLUDE_EVENTS

This macro includes support for Events. The default value is 0 (disables the feature).

UASDK_INCLUDE_ALARMS_AND_CONDITIONS

This macro includes support for Alarms and conditions. The default value is 0 (disables the feature)

UASDK_INCLUDE_HISTORY_ACCESS

This macro includes History Access support. Defualt value is 0 (disables the feature)

UASDK_INCLUDE_AGGREGATES

This macro includes aggregates support. Default value is 0 (disables the feature)

UASDK_USE_HISTORY_ACCESS_HELPERS

This macro includes history access helpers support. Default value is 1.

Certificate Management

Below are the macros which are required for certificate management helpers.

UASDK_INCLUDE_CERTIFICATE_MANAGEMENT_PULL_HELPER

This macro includes support for certificate management pull helper. Default value is 0.

UASDK_INCLUDE_CERTIFICATE_MANAGEMENT_PUSH_HELPER

This macro includes support for certificate management push helper. Default value is 0.

UASDK_INCLUDE_PRIVATE_KEY_REQUEST_FROM_GDS

This macro includes support for private key request from GDS. Default value is 0.

SDK Distribution and Examples

Below section describes the components that SDK package has and all the examples are provided.

Distribution Organization

The SDK is delivered in a distribution with the following folder structure.

Distribution

- Examples
  - Libraries
  - NodesetFiles
  - Linux
    - MAKEPP_multi_threaded
    - MAKEPP_multi_threaded_for_ctt
    - MAKEPP_single_threaded
    - MAKEPP_micro_embedded_device_server
    - MAKEPP_nano_embedded_device_server
    - MAKEPP_events_alarms_and_conditions_mt
    - MAKEPP_client_example_mt
    - MAKEPP_history_console_client
    - MAKEPP_history_sqlite_mbedtls_mt
  - VS2015
    - mbedtls_and_multithreading
    - mbedtls_and_multithreading_for_ctt
    - no_security_single_threaded
    - micro_embedded_device_server
    - nano_embedded_device_server
    - events_alarms_and_conditions_mbedtls_mt
    - console_client
    - history_sqlite_mbedtls_mt
    - history_console_client
  - C_API
- SDK Documentation
  - API Reference
- SDK_Source
  - interface
  - internals

Examples

The desktop is a highly productive environment to experiment with the SDK and learn how to use the API. It is recommended to start by familiarizing with the SDK using the native Linux or Windows examples. The examples directory contains a variety of examples that allows the user to develop with the SDK.

Linux Examples

MAKEPP makefile based projects are provided to facilitate simple makefile based development on x86.

Follow the steps below to configure the development environment and to use the Linux examples out of the box where a Linux system is not available:

1. Install 32-bit or 64-bit Ubuntu Desktop 14.04 LTS into a Virtual box or VMware Virtual Machine.
2. Update the package manager with "sudo apt-get update".
3. Install G++ with "sudo apt-get install build-essential".
4. Install OpenSSL with "sudo apt-get install openssl libssl-dev".
5. Install a Java Runtime with "sudo apt-get install default-jre".
6. Install makepp with "sudo apt-get install makepp"
7. Open terminal in opcua_example folder path and use **makepp** to build the example.

Server Examples:

- A multi-threaded configuration with events and alarms and conditions support
- A multi-threaded configuration with security support using OpenSSL and XML nodeset import using TinyXML2
- A multi-threaded configuration with security support using OpenSSL and XML nodeset import using TinyXML2 for CTT (tested with version V1.2.336.273 and V1.3.340.380)
- A multi-threaded configuration with history support using SQLite3 database, security support using OpenSSL and XML nodeset import using TinyXML2
- A single-threaded configuration with no security and XML nodeset import using TinyXML2
- A single-threaded configuration configured for a small footprint (Micro Embedded Device profile) with no security, nodeset import from ROM and a dedicated memory allocator
- A single-threaded configuration configured for a small footprint (Nano Embedded Device profile) with no security, nodeset import from ROM and a dedicated memory allocator

Client Examples:

- A multi-threaded console client
- A multi-threaded history console client

To build the SDK library and example project change directory to "Distribution/Examples/Linux/MAKEPP_* / opcua_client_example" and execute "makepp clean".

Both the makefile and Eclipse CDT projects can be easily modified to build the library and a sample project for embedded Linux environments by changing the compiler names to the appropriate cross-compiler names and adding any compiler specific and linker flags that the platform requires.

Visual Studio 2015 (VS2015)

Visual Studio 2015 solutions are provided for developing in a Windows environment.

Server Examples:

- A multi-threaded configuration with events and alarms and conditions support

- A multi-threaded configuration with security support using OpenSSL and XML nodeset import using TinyXML2
- A multi-threaded configuration with security support using OpenSSL and XML nodeset import using TinyXML2 for CTT (tested with version V1.2.336.273 and version V 1.3.340.380)
- A multi-threaded configuration with history support using SQLite3 database, security support using OpenSSL and XML nodeset import using TinyXML2
- A single-threaded configuration with no security and XML nodeset import using TinyXML2
- A single-threaded configuration configured for a small footprint (Micro Embedded Device profile) with no security, nodeset import from ROM and a dedicated memory allocator
- A single-threaded configuration configured for a small footprint (Nano Embedded Device profile) with no security, nodeset import from ROM and a dedicated memory allocator

Client Examples:

- A multi-threaded console client
- A multi-threaded history consoled client

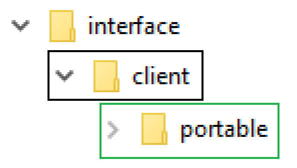
SDK Documentation and Source

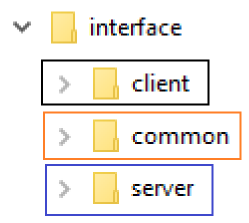
The SDK documentation folder contains the following documents:

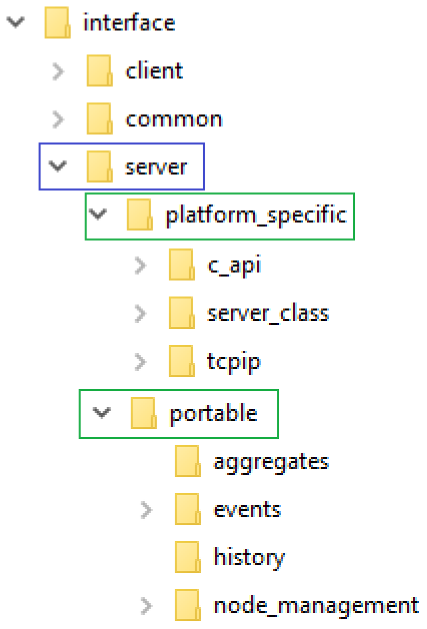
- A user's manual in PDF format
- End User License Agreement in PDF format
- Release Notes in PDF format
- A Security Considerations document in PDF format
- Document detailing changes made to the APIs since the previous release in PDF format
- An API reference manual in HTML format located at "<installed path>/SDK Documentation/API Reference/index.html".

The SDK source folder contains the SDK interface and the SDK internals in obfuscated source code format.

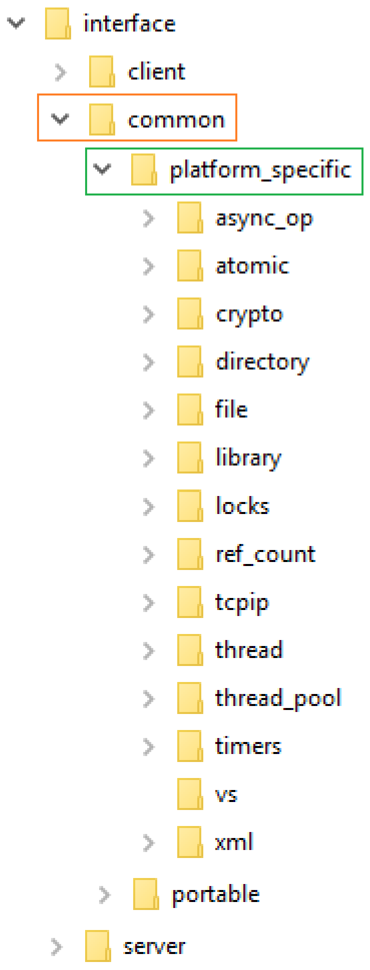
The folder structure of interface folder is as follows:



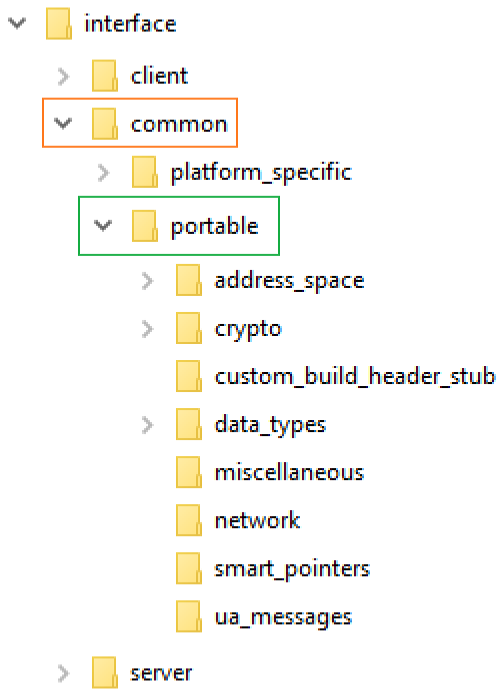




```
interface
  client
  common
  server
    platform_specific
      c_api
      server_class
      tcpip
    portable
      aggregates
      events
      history
      node_management
```



```
interface
  client
  common
    platform_specific
      async_op
      atomic
      crypto
      directory
      file
      library
      locks
      ref_count
      tcpip
      thread
      thread_pool
      timers
      vs
      xml
    portable
  server
```



```
interface
  client
  common
    platform_specific
    portable
      address_space
      crypto
      custom_build_header_stub
      data_types
      miscellaneous
      network
      smart_pointers
      ua_messages
  server
```

Creating an OPC UA Server

OPC UA server is the application which allows user to expose the required data value to the client application. How to create OPC UA Server application is described below.

Creating Server

The easiest way to get started is to take one of the examples provided and use that as a basis for development. Follow the below steps for building the server:

1. Select the example of the user’s choice from the SDK package, based on the Linux/Windows platform.
2. Select the appropriate build configuration, and build the example.

   Once the build is completed without any errors, server is ready to run.

3. Run the server and wait for a message (Discovery and Server endpoints are at: 'opc.tcp://localhost:55000') to pint on the console.
4. Select the OPC UA Client of the user’s choice and connect to the server, once the message is printed.

Note:

If the server is running in different machine or in VM of same machine, ensure to replace the **localhost** with **ip address** of machine or VM where server is running.

Configuring SDK

SDK can be configured to achieve the user needs. In this section describes how to configure the SDK for Server and/or client application.

Build Configuration

UASDK_INCLUDE_SERVER

This macro is used to include server functionality. Set the value to 0 to exclude and 1 to include. The default value is 0.

Configuring for Backward Compatibility (C API)

Set the below macros for using the modules in Legacy C API based server application.

- **UASDK_INCLUDE_SERVER_C_API**

Runtime Configuration

SDK build configuration is controlled through build macros defined in uasdk_default_build_config.h. In addition to the build configuration, the SDK must be configured at runtime by passing in a configuration data object to Initialize () implementation of IServerCore_t module. The configuration data structure is very important as it places limits on what the server allows a client to do. These limits are essential in preventing the out of memory conditions at runtime. Other limits prevent operations that may result in very high CPU utilization.

Additionally, if the user wants to certify the server for compliance with the standard, the server will need to confirm to a defined profile or a superset of that profile. Depending on the requirements this may mandate support for certain configurations settings.

The server can be configured by implementing the IServerConfiguration_t module.

Refer to the **Runtime Configuration APIs** section for configuration related APIs.

XML Import

XML Import functionality can be provided by the third-party libraries TinyXML2 or LibXML2. The TinyXML2 implementation required is a modified one which is provided in the common platform layer. LibXML2 is not provided and the library must be obtained independently and linked with the SDK. The appropriate library can be selected in the build configuration.

The main functional difference between both libraries is that TinyXML2 does not support the validation of the nodeset file against the schema whereas LibXML2 does. The advantage of TinyXML2 is that it has a small footprint (as its name suggests) and is highly portable.

XML parsing requires a lot of memory and CPU bandwidth, so may not be suitable for every platform.

Importing a Nodeset File

The import () function is used to populate an address space using the input nodeset file.

If a schema is provided and if the xml parser library function supports the validation of xml file against a schema, then Import () function will first validate the xml file against the schema. If validation against the schema passes without any error, nodes present in the nodeset file will be created in the address space.

Reading and Writing Variables

After configuring a server and populating the address space, the next step is to connect the variables in the server to the application data sources.

There are four ways to do this:

Data Storage Directly in Variable Nodes

If the data does not change very often or changes in an event driven manner, it may be preferable to simply store the data in the nodes themselves. This is done by creating a DataValue_t (which contains the value, timestamps and quality) and placing it in the Value attribute of the corresponding variable node in the address space. The read service and the monitored items in the server will always use a stored value if one is present when they are reading or sampling the value attribute of a node.

This is a simple approach but has some limitations. For example, if there is a lot of data that changes occasionally, then the values must be updated often even if no client is interested in them.

Implementing the INodeValueAttributeReaderWriter_t Interface

If reader/writer needs to be deferred until they are required, then implement this interface and register the reader/writer with the node. There can be a separate reader/writer for each node or alternatively there could be a single global reader/writer for all nodes or a reader/writer per data source.

Synchronous Access

Synchronous access is the preferred method of data access with the reader/writer. In the case of fast data source, e.g.: Data stored or cached in local RAM, returns the value immediately to the caller. This is the preferred simple programming model which should be implemented as much as possible.

Asynchronous Access

If the user is accessing a slow data source, then it is not desirable to block the call while reading or writing the data. If the user has a single-threaded implementation, all other SDK behavior must wait during the read which may be non-workable. For example, if the data access takes 10ms and the accesses cannot be optimized or aggregated in any way. If a client creates 100 monitored items, then all the CPU time would be spent waiting for the data to be read and the server would not function. In this case, asynchronous access must be performed where the application accesses the data source asynchronously and provides the result to the server when it is ready.

In a multi-threaded configuration, any thread can complete the asynchronous operation by calling the completion object. In a single-threaded configuration, the completion object must be called on the SDK thread even if the data source access takes place on another thread. On a bare metal platform, the data source IO can be interleaved with server operation in the main loop.

Warning:

When using asynchronous data source IO, the application is responsible for completing all operations. Failing to do so will result in memory leaks and incorrect server behavior.

Events, Alarms and Conditions

The Matrikon® Flex OPC UA SDK supports the publishing of event notifications. The events system is divided into two parts.

Event support covers audit and system events that are generated primarily by the SDK stack during server operations. Events can be enabled by defining the **UASDK_INCLUDE_EVENTS** =1 build macro.

Alarms and Conditions support extends the events system to provide support for alarms, acknowledgement, and related operations as defined by the OPC UA Specification. Alarms and conditions support is enabled by defining **UASDK_INCLUDE_ALARMS_CONDITIONS** = 1. Events must be enabled to use Alarms and Conditions.

Events Subscription

Events, Alarms, and Conditions when triggered are delivered as event notifications to the OPC UA client. The OPC UA Client must create one or more monitored item of the event type indicating which address space object or view they wish to monitor for events. When the Events system is enabled, the Server node is marked as an event notifier node by setting the EventNotifier property true on the node.

Additional EventNotifier objects and views can be added to the address space to allow the OPC UA client to filter the events as per the address space the event originates from. By default, only the Server object will be configured to be an event notifier. To add additional event notifiers:

Set the server configuration option IsServerObjectOnlyEventNotifierNode to false.

Add or modify the objects and views that are intended to be used as event notifiers. The EventNotifier field of the object or view must be set to true, and the HasEventNotifier reference type must be added hierarchically to link the server node to each event notifier or tree of notifiers below it.

Warning:

1. Using additional event notifier nodes will make dispatching event notifications utilize more processor time. It is only recommended for non-embedded systems.
2. In Server configuration make sure to set MaxMonItemQueueSize() close to the number of event notifications. For Ex: If number of events generated is 10, then MaxMonItemQueueSize() should be greater or equal to 10. If this value is set to low, then some even notifications will be lost.

Generating Events

While most events are sent automatically from the stack, it is possible to manually send an event. Follow the below steps to manually send an event:

1. Create the event data object and populate the fields.
2. Obtain the node ID of the source node (IntrusivePtr_t < NodeId_t >).

   If there is no specific source in the address space, use the server node (numeric node id, namespace 0, identifier OpcUaId_Server).

3. Obtain or generate a date/time stamp when the event occurred.
4. Call FireEvent (event data, source node, timestamp) in the UAServer_t instance.

Note:

Do not use this method to send event notifications for Alarms and Conditions.

Logging Audit Events

Audit events generated by the SDK server stack can be captured and logged by the application. An implementation of IAuditEventListener_t must be implemented, and registered with the UAServer_t object via RegisterAuditEventListener.

Configuring Events

The server configuration object provides the EventTypeActivationStatus field to allow the developer to select the type of audit and system generated events to be generated.

Alarms and Conditions

Alarms and conditions expand upon the events system and provide a model to manage alarms and conditions present in the system that the software is connected to. To implement an alarm or condition:

Identify the modelling object that best matches the use case. Any of the types derived from BaseCondition_t can be used.

Create and configure an instance of the type. The condition ID must be unique both among conditions, and must be unique in the address space.

Attach listeners to the condition object. Listeners allow the application to observe when the UA client makes method calls on the condition, and respond accordingly.

Register the condition with the ConditionRegistrar. The ConditionRegistrar is exposed as a member of the ServerConfiguration object.

Optionally, mount the condition in the address space. The *BaseCondition_t::MountInAddressSpace()* method is used to expose the condition object modelled in the address space. This allows the current state to be queried by an OPC UA Client using the Read and MonitoredItem services.

Warning:

⚠ Mounting a condition in the address space generates several nodes in the address space and uses a large amount of memory. It is not recommended for embedded systems.

Use the *TransitionTo* methods provided by each type to change the state of the condition and trigger an event notification to trigger an event notification on alarm or condition.

For example, *AlarmCondition_t* derived types, uses the *TransitionToActiveState* and *TransitionToInactiveState* to update the *ActiveState* field and trigger event notifications.

Enable/Disable Server Application to Receive Event Notifications

The server configuration object provides the ***IsServerApplicationReceivingEventNotificationEnabled*** field to allow the developer to enable/disable server application receiving event notifications.

When this option is disabled, all event notifications generated by the server are dumped within the server stack instead of sending it to the UA Client.

Callbacks/Listeners

The server can get the notification on any change in the monitored items or registering of nodes or method call etc. With these callbacks, the server developer can perform necessary actions on or before responding to requests from the client.

Some of these callbacks/listeners can be set at the of run time configuration. Refer to the **Callbacks/Listeners** section for the APIs.

Filters

Monitored items employ filters when sampling data. These filters can include absolute and percent dead band filters. The client specifies what filter it requires when creating a monitored item. Percent dead band filters use floating point operations which may be very slow on platforms that do not have hardware floating point units. Some hardware, such as ARM Cortex-M4F microcontrollers have single precision floating point units but double precision arithmetic must be performed in software. To minimize excessive CPU utilization, it is possible to restrict the use of percent dead band filters to single precision values and to scalar variables.

Refer to the **Filters** section APIs for configuring during runtime.

Profiles

SDK supports 'Embedded' server profile, Micro device server profile, and Nano device server profile. Following APIs can be used to configure the server for different profiles.

Profile configuration can be done at the run time. Refer to the **Profiles** section for the APIs.

Intervals

The server executes at a cyclic rate defined by the application. The SDK must be polled at this cyclic rate. All samplings that the server performs can only be a multiple of the server cyclic rate.

Intervals can be configured during runtime configuration. Refer to the **Intervals** section for the APIs.

The cyclic rate at which the server executes and the rate at which the server is polled by the application. The default value is 100 ms.

⚠ **Warning:**

The faster the cyclic rate the more CPU bandwidth will be consumed by the server. Do not execute the server faster than required. The server will still reply asynchronously to incoming network traffic regardless of the cyclic rate. One second for example would be perfectly acceptable for a server that does not require sampling data at a faster rate than 1. 100ms would be a typical rate and 50ms would be the maximum rate encountered in practice.

Building Server on Different Platform

SDK is portable to any platform which has the minimal hardware requirements like ARM Cortex M4 architecture (Refer to the **FLASH and RAM** section and ensure that the platform in which OPC UA Server needs to be implemented has required memory footprint).

Refer to the **Platform** Requirement section for pre-requisites on building the SDK.

Memory Usage and Statistics and Performance

For embedded platforms memory usage statics is a critical point or sometimes a deciding factor to include application in embedded device. Below Section describes how to generate memory usage statics information for the required application.

Memory Utilization

The heap can be used for SDK memory allocations by modifying the configuration accordingly. In the majority of applications, it is expected that all memory allocations will be performed from the buffer provided as part of the configuration that is passed to the UAServer_Initialize() function.

Set the memory buffer to a very large value or as large a value as is practical on the platform to determine how much memory is required.

1. Run the server for an extended period exercising it in such a way that the maximum memory utilization is performed (considering the runtime configuration limits that has been set).
2. Call the memory info API function to obtain the recommended memory for the test run.

The developer should take note of the following factors to ensure that the SDK uses the maximum memory possible:

- The number of concurrently connected clients is important.
- Ensure that Browse, Read and Write service calls are made by a client that are as large as that expected to be performed by a client in a production environment.
- Create as many monitored items as the server can support. Where some nodes have larger data payloads than others (arrays for example) ensure that the monitored items created reflect the expected behavior in production. The bigger the data payload, the more memory each monitored item will require.

  Where the server configuration allows for monitored item sample queue lengths greater than 1, configure subscriptions such that the monitored item will be queuing the maximum number of samples before reporting samples to the client. For example, where the maximum server sample queue length is 10, sample the monitored items at a rate > 1/10th of the subscription reporting rate to maximize the internal sample buffering in the server.

Warning:

- Running the server with less than the recommended amount of memory can result in memory exhaustion during server operation which will result in undesirable behavior such as clients being forcibly disconnected from the server, monitored item sampling failure amongst other behaviors etc.
- Calculating the recommended memory after running a test which does not sufficiently stress the server memory subsystem, will not yield a useful result and the memory recommendation reported by the function cannot be relied upon.

FLASH and RAM

These are the minimum ROM and RAM requirements for the OPC UA Server SDK, excluding the resources required by the device application and the TCP/IP stacks. Depending on the functional requirements of an application in terms of address space contents, several concurrent connections, monitored items, etc., the RAM required can be much higher.

Refer to the following table for the Minimum Server Footprint:

Profile Configuration	**Flash (kB)***
Nano Embedded Device Server	666
Micro Embedded Device Server (4 Monitored items)	755
Embedded Server (including security and full address space and 10 Monitored Items)	917
Embedded Server (including security and full address space and 100 monitored items)	917

*Metrics obtained for ARM Thumb2 instruction set (Cortex-M7F), Atollic TrueSTUDIO for Linux 9.2.0, GCC –Os -flto

Table 3 - Minimum Server Footprint

The Flash footprint is given for an ARM Cortex-M4 microcontroller (Thumb 2 instruction set). Flash footprint can vary dramatically from one architecture to another. Hence these metrics must be viewed in the context of the test architecture.

RAM usage varies greatly depending on the required feature set. The RAM requirement for the SDK begins at a few 100kB.

The minimum recommended microcontroller class for a Micro Embedded Device Server with a very small address space and a small number of monitored items would be a 50MHz ARM Cortex-M3 MCU with 1MB Flash and an external SRAM IC.

The minimum recommended microcontroller class for an Embedded Server (including security) with a very small address space and a small number of monitored items would be a 100MHz ARM Cortex-M3 MCU with 2MB Flash and an external SRAM IC.

Performance Issues

As discussed previously all callbacks must be completed quickly to provide good server performance and low latency operation. System calls made in Linux and other OS's during callback operations may cause a context switch and can dramatically reduce the performance of the application. Where large data sets are being monitored or where efficient CPU utilization is important, all steps must be taken to ensure efficient callback implementations.

Performance can be improved by optimizing the SDK for speed using the relevant builds macro although this can significantly increase memory usage depending on the max message and max message chunk size configuration.

The performance of address space node lookups can be improved by ensuring that SDK hash tables use a bucket count which is a prime number. This minimizes the collisions when searching the tables, thereby improving performance.

Refer to the following table for typical server performance:

Test	**Conditions**	**Hardware**	**CPU Utilization (%)**
100 continuously changing tags	Sampling and reporting every 100ms	ARM Cortex-M4F (STM32F407) @ 168MHz	16%

*Metrics obtained using GCC -O3

Table 4 – Typical Server Performance

Available Features in SDK

SDK performs some of the below mentioned features to support the Server Application Developer:

- Attribute Changed Listener
- Attribute About to be Read Listener
- Address Space Listener
- Server State Listener

Attribute Changed Listener

Attribute changed listener is the module which can be set for any type of node. If the client has changed the value of the attribute, SDK will give a callback to the application layer. This feature can be used to take some custom action once specific node's attribute has changed.

Perform the following steps to use this feature:

1. Implement a class which is inherited from **INodeAttributeChangedListener_t**.
2. Ensure to include attributes that is compatible to the **changed listener**.
3. Create an object of this derived class, and use the attribute **changed listener** with the help of **AttributeChangedListenerAdd()** for the required node.

Attribute About To Be Read Listener

Attribute about to be read listener is the module which can be set for any type of node. If the client is about to read value of the attribute, SDK will give a callback to the application layer. This feature can be used to take necessary action before the node's attribute value is read.

Perform the following steps to use this feature:

1. Implement a class which is inherited from **INodeAttributeAboutToBeReadListener_t**.

2. Ensure to include attributes that is compatible to the **about to be read listener**.
3. Create an object of this derived class, and use the attribute **about to be changed listener** with the help of **SetAttributeAboutToBeReadListener()** for the required node.

Address Space Listener

Address space listener is the module which can be set for any change in address space by adding or deleting node to/from address space. SDK will give a callback to application layer. This feature can be used to take the necessary action when a specific node gets added/deleted to/from address space.

Perform the following steps to use this feature:

1. Implement a class which is inherited from **IAddressSpaceListener_t**.
2. Ensure to include attributes that is compatible to the **address space listener**.
3. Create an object of this derived class, and use the address space listener with the help of **AddressSpaceListenerAdd()** on the address space.

Server State Listener

Server State Listener is a module which can be set for any change in server state; SDK will give a callback to application layer. This feature can be used to take the necessary action when state of the server is changed.

Perform the following steps to use this feature:

1. Implement a class which is inherited from **IServerStateListener_t**.
2. Create an object of this derived class, and assign the server state listener with the help of **ServerStateListener()** on the server configuration.

Service Call Listener

Service call listener is a module can be implemented by the Application layer to take certain action before SDK process the service request.

Perform the following steps to use this feature:

1. Implement a class which is inherited from **IServiceCallListener_t**.
2. Create an object of this derived class, and assign the service call listener with the help of **ServiceCallListener()** on the server configuration.

Secure Channel Listener

Secure Channel listener is a module which can be set for open and/or close of secure channel. SDK will give a callback to application layer if channel listener is set during server configuration. This feature can be used to take necessary action when secure channel is opened and/or closed.

Perform the following steps to use this feature:

1. Implement a class which is inherited from **IChannelListener_t**.
2. Create an object of this derived class, and assign the channel listener with the help of **ChannelListener()** on the server configuration.

Miscellaneous

Few modules are implemented in SDK for internal purposes and are critical for SDK to function. Implementation is included in SDK, but the interface files are provided so that an SDK user can write their own version of the required modules.

Refer to the

Other Modules in Server section for the APIs.

Registering Server to Discovery Server

The discovery process allows Clients to find Servers on the network. *Clients* and *Servers* can be on the same host, on different hosts in the same subnet, or even on completely different locations in an administrative domain.

Currently SDK supports only Local Discovery Server which is running on the same machine where the server is running.

Refer to the Macros to include and APIs that can be used to discover the server in **Registering Server to Discovery Server** appendix section.

Note:

Servers created using SDK are tested against Local Discovery Server given by the OPC Foundation.

Server Diagnostics

Server created using SDK can enable the server diagnostics feature.

Refer to the build macros for enabling the server diagnostics in **Server Diagnostics** appendix section.

Note:

Ensure to set the ServerDiagnosticsEnabledFlag to true with the help of API **ServerDiagnosticsEnabledFlag()** in the server configuration.

Address Space Roles and Permissions

Introduction

The FLEX SDK predates the addition of roles and permissions functionality to the OPC UA specification. This functionality was added to OPC UA with version 1.04, released in 2017. OPC UA roles and permissions are now fully supported by the SDK and it is recommended that this feature set is used. In general, usage is optional and application developers are free to continue to use the legacy user permissions functionality. When implementing Pub/Sub however roles and permissions must be used as this is a required dependency for Pub/Sub in the FLEX SDK.

Roles

A role in OPC UA is analogous to a group of users that are authorized to perform various activities in the server. Example roles would be operator, engineer and administrator. The specification defines some default, well-known roles that application developers can use. These well-known roles are sufficient for many use cases but if you have more specific requirements you can also define your own roles.

Permissions

Permissions define what a particular role can do in relation to the server address space. Example behaviours are browsing nodes, reading and writing node attributes and calling methods. Permissions are assigned to roles. A role and its associated permissions are called a RolePermission, that is, a role and its permissions.

RolePermissions

RolePermissions can be set for all the nodes in a namespace or for specific nodes. The namespace metadata contains the default role permissions for the nodes in that namespace. The role permissions for a namespace is an array of individual RolePermissions, as several roles may have permissions within that namespace. These role permissions apply to any nodes in that namespace that do not have node specific role permissions defined. Where a node has node specific role permissions assigned then these override any default role permissions defined for the namespace.

Assigning Roles

Roles are assigned to a session whena client activates the session within the server. A single session can have multiple assigned roles. For example, a single user identity may map to operator, ConfigureAdmin and SecurityAdmin roles. Roles are assigned not only based on session or user identity but also based on the server endpoint along with the client application identity itself as defined by the application URI. For example, a given role may only be assignable when a client connects over a specific network interface, or to a specific server endpoint. A role may also only be assignable to a specific client application instance.

Calculating Permissions

When a client attempts to interact with a server via service calls, the SDK will dynamically calculate the allowed permissions for the session for each requested operation. The session permissions for a given node are the sum of the permissions for that node, for all roles assigned to the session.

RoleSet Information Model

The server address space contains a RoleSet object that contains the list of all possible roles in the server. If the server implementer allows, a client with suitable permissions can also dynamically configure roles and their associated permissions via the server address space.

Note:

Great care must be taken when editing the roles and permissions settings remotely via the information model.

Session roles are dynamically recalculated upon receipt of every service call so any changes made will take immediate effect. If you are not careful you can easily downgrade the permissions of your session in real time or even lock yourself out fo the server.

Enabling Roles and Permissions

According to the OPC UA specification, the provision of namespace metadata is optional. In contrast, the FLEX SDK roles and permission implementation requires all namespaces in a server address space to have a namespace metadata object defined. This means that role permissions cannot be enabled on a per-namespace basis. Once a server application is built with this feature it is applied globally to all namespaces in all servers in the application and all namespaces must support the feature.

To enable roles and permissions:

- Build the SDK with UASDK_INCLUDE_ROLE_PERMISSIONS set to 1.
- Load an information model into the server address space that includes the well-known RoleSet object and the well-known roles.
- Take the UserPermissionsProvider_t implementation from one of the example projects provided that support roles and permissions and use this as as a starting point for your application. Modify the class as required.
- Immediately after server initialization, call ***UserPermissionsProvider_t:: InitialiseRolesAndPermissionsInAddressSpace()*** to initialize the feature set.
- Create any required users and add the relevant identity mapping rules to map the users to roles.
- Apply default role permissions to the namespace metadata objects of each namespace in the address space.
- Apply role permissions to nodes in the address space as required by populating the RolePermissions attribute of the node.

Role and Permissions Related Classes

RoleSet_t

Represents the RoleSet object referenced by the ServerCapabilities object. Automatically created on server, publisher or subscriber initiailsation and available from ***ICommonConfiguration_t::Roleset()***.

INamespaceMetadataHelper_t

Makes it easy to create and lookup the NamespaceMetadata objects for all namespaces. Automatically created on server, publisher or subscriber initiailsation and available from ***ICommonConfiguration_t::NamespaceMetadataHelper ()***.

UserPermissionsBase_t

Implements many aspects of roles and permissions permissions lookup as an assistance to the application developer. Should be subclassed to create a concrete implementation of **IUserPermissions_t**. Examples implementations are provided.

Role_t

Represents a role as exposed as a child of the RoleSet object.

RolePermissionType_t

A mapping of a specific role to a defined set of permissions.

PermissionType_t

A bitmap of permissions defined for a specific role or set of roles for a namespace or a specific node.

Node Management Service

Introduction

The Matrikon FLEX OPC UA SDK supports node management services. Based on the requirement, you can use this to modify the information model or address space of the server. The implementation details and how to enable the NodeManagement services are described below.

For the default implementation provided by SDK, navigate to <Path to SDK_Source>/interface/server/portable/node_management/. Considering the default implementation as an example, the server application developer can modify or rewrite the implementation based on the requirement.

To support the node management, make sure that build macro UASDK_INCLUDE_NODE_MANAGEMENT_SERVICE_SET is enabled by setting its value to 1.

Server-side implementation

In the default implementation, all of the Node Management services should initialize with the thread pool object. Based on the requirement, the server application developer can either create a new thread pool to handle the node management alone or can use the same thread pool associated with the server instance.

Add Nodes service

To use the default implementation of an add nodes service the server application layer should create an instance of `AddNodesRequestHandler_t` class and initialize the object with the following:

- Whether to allow the service to create a node with the node id already existing in address space.

- Whether to create an optional child node from the type definition modelling rule.
- Thread pool object.
- Address space in which nodes should be added.

Once the `AddNodesRequestHandler_t` object is created and initialized, it is recommended to configure the same in the `IServerConfigutation_t` object using `AddNodesRequestHandler()` API.

Working

When Server SDK receives Add Nodes service request, it will check for the AddNodesRequestHandler_t object in the IServerConfiguration_t object and calls the API BeginAddNodes() with the necessary information. The AddNodesRequestHandler_t will create AddNodesServiceCallWorker_t (IRunnable_t) instance and queue this IRunnable_t object to thread pool initialized with AddNodesRequestHandler_t. When this specific AddNodesServiceCallWorker_t object gets a change to complete its job from the thread pool, it will check for the correctness of the data provided with the Add Nodes service call and add the node in the address space. If the type definition has the modeling rule with child nodes, the child nodes will be created in the instance creation of the node.

Delete Nodes service

To use the default implementation of the delete nodes service, the server application layer should create an instance of `DeleteNodesRequestHandler_t` class and initialize the object with the following:

- Thread pool object.
- Address space in which nodes should be added.

Once the `DeleteNodesRequestHandler_t` object is created and initialized, it is recommended to configure the same in the `IServerConfigutation_t` object using `DeleteNodesRequestHandler()` API.

Working

When Server SDK receives Delete Nodes service request, it will check for the DeleteNodesRequestHandler_t object in the IServerConfiguration_t object and calls the API BeginDeleteNodes() with the necessary information.

The DeleteNodesRequestHandler_t will create DeleteNodesServiceCallWorker_t(IRunnable_t) instance and queue this IRunnable_t object to thread pool initialized with DeleteNodesRequestHandler_t. When this specific DeleteNodesServiceCallWorker_t object gets a change to complete its job from the thread pool, it will check for the correctness of the data provided with the Delete Nodes service call and delete the node from the address space.

Add References service

To use the default implementation of an add references service, the server application layer should create an instance of `AddReferencesRequestHandler_t` class and initialize the object with the following:

- Thread pool object.
- Address space in which nodes should be added.

Once the `AddReferencesRequestHandler_t` object is created and initialized, it is recommended to configure the same in the `IServerConfigutation_t` object using `AddReferencesRequestHandler()` API.

Working

When Server SDK receives Add References service request, it will check for the AddReferencesRequestHandler_t object in the IServerConfiguration_t object and calls the API BeginAddReferences() with the necessary information.

The AddReferencesRequestHandler_t will create AddReferencesServiceCallWorker_t(IRunnable_t) instance and queue this IRunnable_t object to thread pool initialized with AddReferencesRequestHandler_t. When this specific AddReferencesServiceCallWorker_t object gets a change to complete its job from the thread pool, it will check for the correctness of the data provided with the Add References service call and adds the references.

Delete References service

To use the default implementation of the delete references service, the server application layer should create an instance of `DeleteReferencesRequestHandler_t` class and initialize the object with the following:

- Thread pool object.
- Address space in which nodes should be added.

Once the `DeleteReferencesRequestHandler_t` object is created and initialized, it is recommended to configure the same in the `IServerConfigutation_t` object using `DeleteReferencesRequestHandler()` API.

Working

When Server SDK receives Delete References service request, it will check for the DeleteReferencesRequestHandler_t object in the IServerConfiguration_t object and calls the API BeginDeleteReferences() with the necessary information.

The DeleteReferencesRequestHandler_t will create DeleteReferencesServiceCallWorker_t(IRunnable_t) instance and queue this IRunnable_t object to thread pool initialized with DeleteReferencesRequestHandler_t. When this specific DeleteReferencesServiceCallWorker_t object gets a change to complete its job from the thread pool, it will check for the correctness of the data provided with the Delete References service call and delete the references.

Server Access Type

OPC UA Specification provides four different types of access:

- Data Access
- Events Alarm
- Conditions and Historical Access
- History Access

Each access types are described below.

Data Access

Data Access (DA) variable types are structured variable type with more features than *BaseDataVariables*. It is valid to use only *BaseDataVariables* to expose the entire data if it is a required functionality. DA types are only necessary when the user needs the extra features that the types provide.

In addition, DA types require more storage in the server hence it is recommended to expose fewer DA types than *BaseDataVariables* for a specific amount of RAM. If the environment is highly resource constrained, use only DA types when it is required. Each DA variable type is described below.

DataItem:

A *DataItem* is the basic DA variable type that inherits from *BaseDataVariable*. All other DA variable types inherit from *DataItem* and have the same properties as a *DataItem* in addition to their own properties. A *DataItem* has two properties:

- ***Definition*** (optional).

  This optional property is a string that defines how the value of the variable is calculated. This string is non-localized (i.e., it is always the same regardless of the language settings for a session[1]) and will often contain an equation that can be parsed by certain clients.

  Example: Definition: = "(TempA – 25) + TempB"

- ***ValuePrecision*** (optional).

  This optional property is a double precision floating point value that specifies the maximum precision that the server can maintain for the item. The *ValuePrecision* provides guidance to the client. A server is required to round any value with more precision than it supports.

[1]Server localization is the creation of a server that exposes its human readable strings in a variety of languages depending on the requirements of a particular client. Localization is supported by the Device Server SDKs but is optional and can be provided on need basis.

For float and double data types, *ValuePrecision* specifies the number of digits after the decimal place. For *DateTime* values, it specifies the minimum time difference in nanoseconds.

AnalogItem:

An *AnalogItem* is a *DataItem* that contains a value that must be of a numerical data type (i.e., not a string or byte array). An *AnalogItem* has the same optional properties as a *DataItem* in addition to the following properties:

- **InstrumentRange** (optional)

  This optional property defines the full-scale range that can be returned by the variable. The property is of type *Range*. The Range type is comprised of two double precision floating point numbers indicating the low and high limits.

- ***EURange*** (mandatory)

  This mandatory property defines the value range likely to be obtained in a normal operation. It is intended for use such as automatically scaling a bar graph display.

- **EngineeringUnits** (optional)

  EngineeringUnits specifies the units for the *AnalogItem's* value (e.g., Hertz, seconds). The *EngineeringUnits* property is comprised of four attributes:

  - ***NamespaceUri**:*
    This string specifies the organization or standard body that defines the engineering units.
  - ***UnitId**:*
    This Int32 is an identifier used for programmatic evaluation by the client. If not used it is set to -1.
  - ***DisplayName**:*
    This localized string is the displayed name of the units and is typically an abbreviation (e.g., "h" for hour or "m" for meter).
  - ***Description**:*
    This localized string contains the full name of the engineering unit.

TwoStateDiscrete:

A *TwoStateDiscrete* is a *DataItem* that contains a Boolean value. A *TwoStateDiscrete* has the same optional properties as a *DataItem* in addition to the following properties:

- ***TrueState*** (mandatory)
  This localized string is the string displayed when the value is TRUE.
- ***FalseState*** (mandatory)
  This localized string is the string displayed when the value is FALSE.

MultiStateDiscrete:

A *MultiStateDiscrete* is a *DataItem* that contains an enumerated value. A *MultiStateDiscrete* has the same optional properties as a *DataItem* in addition to the following property:

- ***EnumStrings*** (mandatory)
  This array of localized strings contains a string to be displayed for each value of the enumeration.

MultiStateValueDiscrete:

A *MultiStateValueDiscrete* is a *DataItem* that contains more than two states and state values do not have to be consecutive numeric values or the enumeration values need not have to start with 0. A MultiStateValueDiscrete has the same optional properties as a *DataItem* in addition to the following properties:

- ***EnumValues*** (mandatory)
  This is an array of *EnumValueType*. Each entry in an array consists of 1 enumeration value with integer notation, a human readable representation and help information.
- ***ValueAsText*** (mandatory)
  This gives human readable text for the value in the *MultiStateValueDiscrete* variable.

ArrayItem:

An *ArrayItem* abstract type is a *DataItem* and it represents the general characteristics of the ArrayItem. The derivative of this type will have the same optional properties as a *DataItem* in addition to the following properties:

- ***InstrumentRange*** (optional)
  This represents the range of values of an *ArrayItem*.
- ***EURange*** (mandatory)
  This represents the range of values of an *ArrayItem* in a normal operation.
- ***EngineeringUnits*** (mandatory)
  This represents the engineering units of values of an *ArrayItem*.
- ***Title*** (mandatory)
  This represents the user-readable title for values of an *ArrayItem*.
- ***AxisScaleType*** (mandatory)
  This represents the scale for the axis where values of *ArrayItem* should be displayed.

YArrayItem:

A *YArrayItem* is an *ArrayItem* and it represents a single dimension array of numeric values used to represent the spectra or distribution where X-Axis intervals are constant. This type will have same optional and mandatory properties of *ArrayItem* and *DataItem* in addition to the following property:

- ***XAxisDefinition*** (mandatory)

  This represents the X-Axis's Engineering Unit and Range.

XYArrayItem:

A *XYArrayItem* is an *ArrayItem* and it represents the vector of *XVType* values like a list of peaks, where *XVType.x* is the position of peak and *XVType.value* is intensity. This type will have the same optional and mandatory properties of *ArrayItem* and *DataItem* in addition to the following property:

- ***XAxisDefinition*** (mandatory)

  This represents the X-Axis's Engineering Unit and Range.

ImageItem:

An *ImageItem* is an *ArrayItem* and it represents a matrix of values like an image, where the pixel position is given by X which is the column and Y, the row. This type will have the same optional and mandatory properties of *ArrayItem* and *DataItem* in addition to the following properties:

- ***XAxisDefinition*** (mandatory)

  This represents the X-Axis's Engineering Unit and Range.

- ***YAxisDefinition*** (mandatory)

  This represents the Y-Axis's Engineering Unit and Range.

CubeItem:

A *CubeItem* is an *ArrayItem* and it represents cube values like a spatial particle distribution where the particle position is given by X which is the column, Y the row and Z the depth. This type will have same optional and mandatory properties of *ArrayItem* and *DataItem* in addition to the following properties:

- ***XAxisDefinition*** (mandatory)

  This represents the X-Axis's Engineering Unit and Range.

- ***YAxisDefinition*** (mandatory)

  This represents the Y-Axis's Engineering Unit and Range.

- ***ZAxisDefinition*** (mandatory)

  This represents the Z-Axis's Engineering Unit and Range.

NDimensionArrayItem:

An *NDimensionArrayItem* is an *ArrayItem* and it represents a generic multi-dimensional *ArrayItem*. This type will have the same optional and mandatory properties of *ArrayItem* and *DataItem* in addition to the following properties:

- ***AxisDefinition*** (mandatory)

  This represents Engineering Unit and Range for all axis.

Warning:

- Any DA node which has instrument range property has a value of data type Complex Number or Dubble Complex Number type, then application developer need to take care of the Range check.
- Change needs to be done in AddressSpaceUtilities_t::RangeCheck() function, address_space_utilities_t.cpp file. It is present in path <path to SDK>/interface/common/portable/address_space/helpers/

Events, Alarm and Conditions

Refer to the **Events, Alarms and Conditions APIs** section for more details.

History Access

Refer to **History Server Example Design and Implementation with Matrikon OPC UA SDK.pdf** document in <path to packages>/flex_R<version_number>/Examples/ for more details.

Creating OPC UA Client

OPC UA client is the application which consumes the data exposed by the server. Steps to create an OPC UA Client application is described below.

Creating Client

The easiest way to get started is to take one of the examples provided and use that as a basis for development. Follow the below steps for building the server:

1. Select the example of your choice from the SDK package based on the Linux/Windows platform.
2. Select the appropriate build configuration, and build the example. Once the build is completed without any errors, client is ready to run.

Build Configuration

UASDK_INCLUDE_CLIENT

This macro is used to include client functionality. Set the value to 0 to exclude and 1 to include. The default value is 0.

Runtime Configuration

SDK build configuration is controlled through build macros defined in uasdk_default_build_config.h. In addition to the build configuration, the SDK must be configured at runtime by passing in a configuration data object to Initialize() implementation of IClientCore_t module. The configuration data structure is very important as it places limits on what the server allows a client to do.

The client can be configured by implementing the IClientConfiguration_t module.

Building Client on Different Platform

The Client part of SDK is a set of cross-platform data types, structures and interfaces to API methods. UA concept can use related objects that the user developers can add OPC UA functionalities to their OPC UA client applications.

Available Features in SDK

This section describes the features which are implemented as a part of Client SDK.

Core Client Characteristics

- Core Client Facet
- Base Client Behaviour Facet
- Discovery Client Facet

- Multi Server Client Connection Facet
- Address Space Lookup Client Facet

Client Data Access

- Attribute Read / Write Client Facet
- Data Change Subscriber Client Facet
- Data Access Client Facet

Generic Client Features

- Method Client Facet
- State Machine Client Facet

Publish Subscribe

Introduction to Pub Sub

OPC UA Client-Server communication requires a unicast TCP connection between a single client and a single server. If a server is to provide data to multiple connected clients, each client must have a dedicated logical connection and make service calls on the server. The server is very limited in its ability to share internal resources amongst connections so in practice the server CPU and RAM utilization increases in direct proportion to the number of clients connected, assuming the clients are performing similar operations.

With PubSub all subscribers can consume the same single stream of data from a publisher. Also, subscribers do not require a direct connection to a publisher. For these reasons PubSub is useful for UA applications that require scalability and/or location independence.

Some typical use cases for PubSub are:

- Peer-to-peer communications between controllers and between controllers and HMIs.
- Asynchronous workflows.
- Logging to multiple systems.
- Streaming data to cloud-based applications for remote condition monitoring, analytics and predictive maintenance.

Entities

A PubSub implementation is composed of a number of entities, some mandatory and some optional. These are described below.

Message Oriented Middleware

The publisher communicates by transmitting messages into a message oriented middleware (MOM). The MOM is an abstraction with a number of different possible implementations. Some example implementations are:

- UADP binary encoded messages over UDP. In this case the network infrastructure is the MOM. This approach is suitable for a LAN environment where efficient transmission of frequent small messages is required.
- JSON encoded messages transmitted to a data broker via the AMQP or MQTT messaging protocols. This approach is suitable for publishing data to cloud applications using cloud native technologies.

PubSub is based on the OPC UA Information Model and typically publishers will be also be servers and subscribers will also be clients. That said, there is no requirement for a publisher to be a server or for a subscriber to be a client. The PubSub information model for configuration allows the creation or consumption of PubSub configuration using the UA Client/Server architecture.

Publisher

A publisher is that which publishes messages to the MOM. A publisher is composed of several stages working together. These stages are described below.

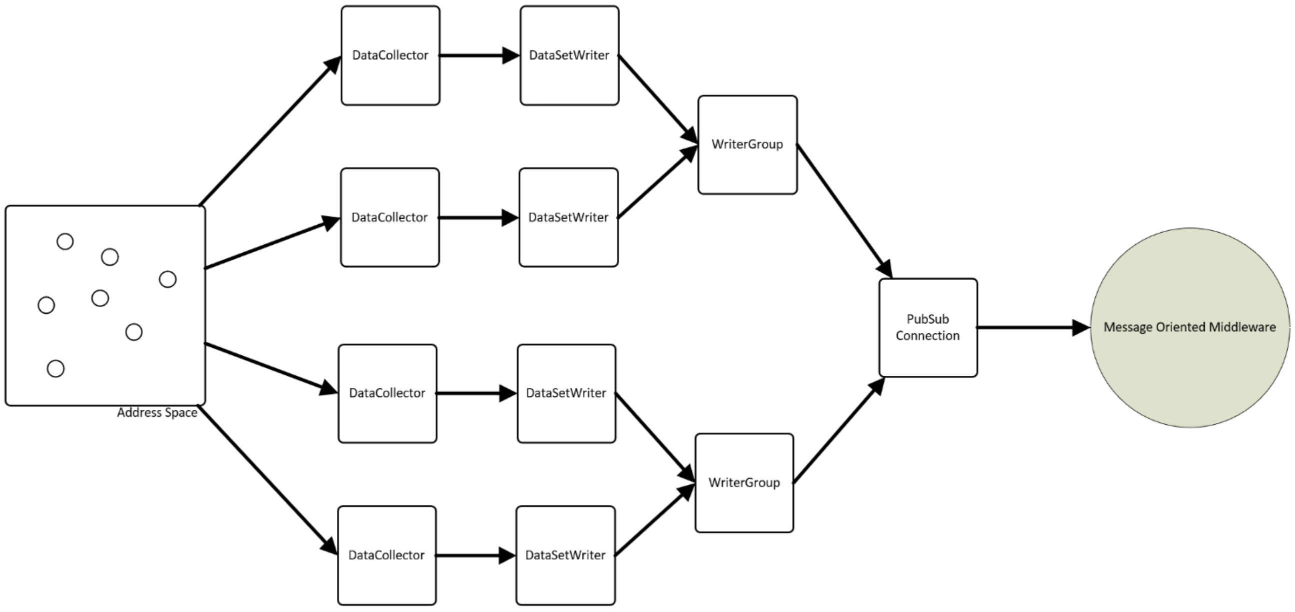


Figure 6 - Publisher Data Flow

Publisher DataCollector

The data collector actively collects data from the publisher information space (address space). Both events and data value changes can be collected. Data changes are collected by periodically sampling the node attributes in the address space in a similar manner to monitored item sampling in OPC UA Client/Server. Events are delivered to the data collector in an event driven fashion where they are buffered for subsequent transmission.

There exists a single data collector for each published data set defined in the PubSub configuration. A published data set is a defined list of values or event fields that will be published in the encoded form of a data set message.

Publisher DataSetWriter

There exists one DataSetWriter for every data collector. A DataSetWriter is responsible for converting a DataSet received from a data collector into a DataSetMessage, every time the publishing interval elapses. The DataSetMessage is the DataSet encoded using the configured message encoding.

A DataSetWriter is considered a child of a WriterGroup. Each DataSetWriter has one parent WriterGroup. When the publishing interval elapses the encoded DataSetMessage is passed to the WriterGroup.

Publisher WriterGroup

A WriterGroup is the parent of one or more DataSetWriters. When the publishing interval elapses each DataSetWriter delivers a DataSetMessage to the WriterGroup and the WriterGroup aggregates these messages into a single NetworkMessage. The NetworkMessage is that which is subsequently transmitted into the MOM. A WriterGroup may also apply message based security to the NetworkMessage.

A WriterGroup is considered a child of a PubSubConnection. Each WriterGroup has one parent PubSubConnection. When the publishing interval elapses the encoded NetworkMessage is passed to the PubSubConnection.

PubSubConnection

A PubSubConnection is the parent of one or more WriterGroups. When the WriterGroups pass NetworkMessages to the connection, the connection transmits these messages into the MOM.

Subscriber

A subscriber is that which subscribers to messages from the MOM. A subscriber is composed of several stages working together. These stages are described below.

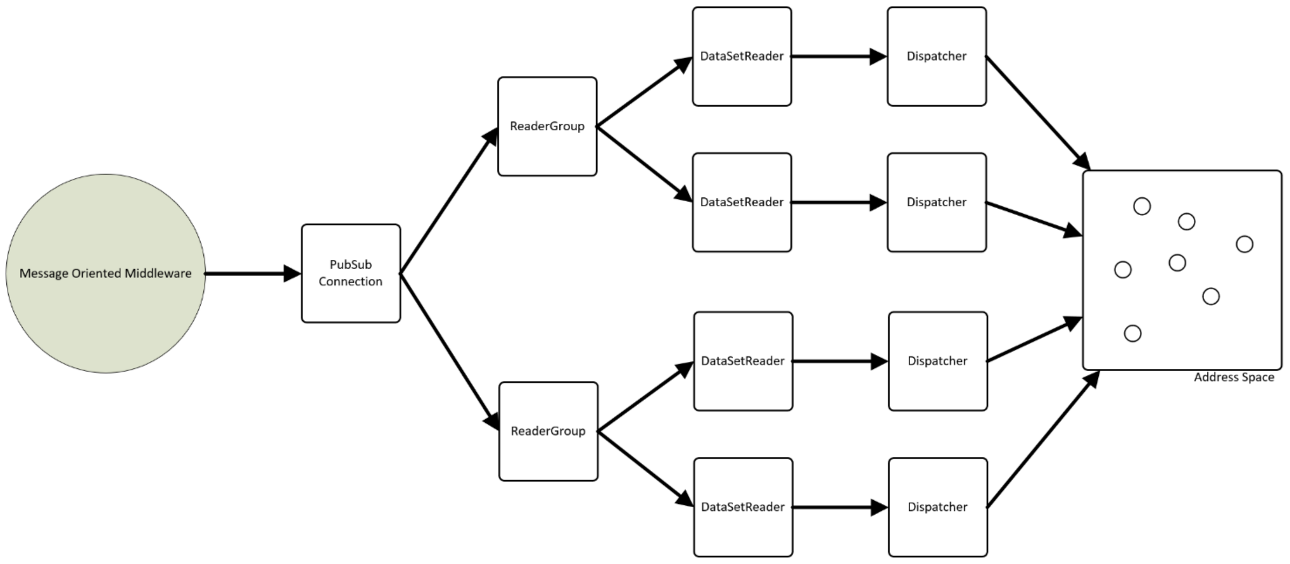


Figure 7 - Subscriber Data Flow

PubSubConnection

A PubSubConnection is the parent of one or more ReaderGroups. When the connection receives network messages it decodes them and passes the DataSetMessages that pass its internal filters to its ReaderGroups.

Subscriber ReaderGroup

A ReaderGroup is the parent of one or more DataSetReaders. When the ReaderGroup receives a set of DataSetMessages it passes them to its DataSetReaders.

Subscriber DataSetReader

When a DataSetReader receives a DataSetMessage it decodes the DataSet and passes it to a Dispatcher for writing to the Subscriber address space. There are two types of dispatcher.

Target Variables Dispatcher

The target variables dispatcher writes the received data to the node attributes defined in the SubscribedDataSet configuration. TargetVariables are used where we want to write the data to pre-existing nodes in the subscriber address space.

SubscribedDataSetMirror Dispatcher

The SubscribedDataSetMirror dispatcher creates a mirror image of the nodes from the PublishedDataSet in the publisher, in the subscriber address space. It then writes the data to the created nodes in the subscriber address space. This dispatcher is used when we want to mirror the publisher source data in the subscriber.

Security Key Service

When using UADP binary message encoding, end-to-end message based security (authentication and encryption) can be applied to the network messages. The Security Key Service is an information model that can be hosted in a server to provide the security keys required to secure NetworkMessages.

The PubSubState State Machine

The PubSubState state machine is a state machine that exists in the components illustrated below. PublishSubscribe is the parent component that contains one or more PubSubConnections. Each connection contains one or more groups and each group contains one or more DataSetReaders or DataSetWriters. Parent state changes affect the state of all children. This behavior is defined in the specification at:

https://reference.opcfoundation.org/v104/Core/docs/Part14/6.2.1/

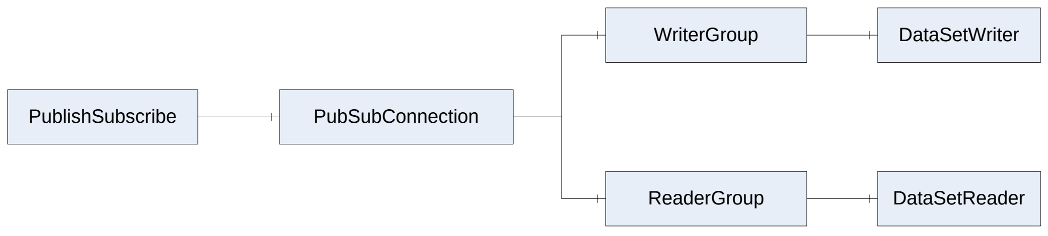
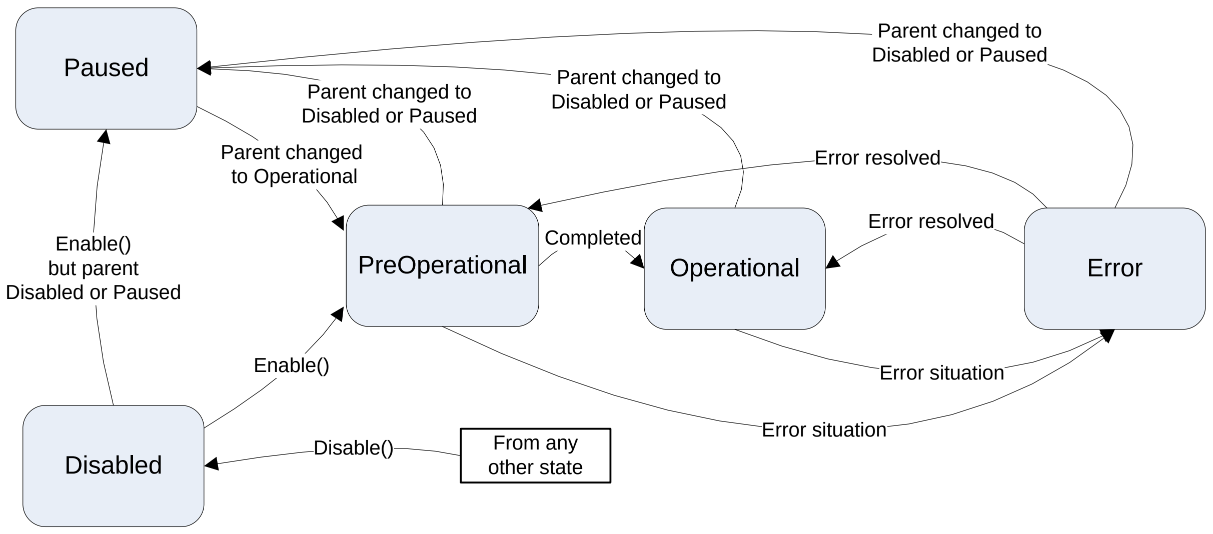


Figure 8 - The components containing a PubSubState state machine and their relationships



Paused
Parent changed to Disabled or Paused
Parent changed to Disabled or Paused
Parent changed to Disabled or Paused
Parent changed to Operational
Error resolved
Error resolved
Enable() but parent Disabled or Paused
PreOperational
Completed
Operational
Error
Enable()
Error situation
Error situation
Disabled
Disable()
From any other state

Figure 9 - PubSubState state machine

Pub Sub Support in the FLEX SDK

The initial PubSub release supports both Publisher and Subscriber with JSON encoded messages over MQTT. The supported message-oriented middleware is any data broker that can send and receive MQTT messages. End-to-end message-based security is not supported in OPC UA PubSub with the JSON encoding but the UA applications can connect to the broker over TLS in order to secure the data, point-to-point, between publisher, broker and subscriber.

Limitations and Considerations

Server Build Dependency

In principle, the Pub/Sub feature is complementary to but independent of Client/Server. Publishers and subscribers can be integrated into the same application as a client and/or a server but there is no absolute requirement to do so. In release R5.0 of the FLEX SDK the PubSub feature set cannot be built without also including the server feature set in the build. This restriction may be removed in a future release.

Threading Model

As with Client/Server, the SDK can be built in a single or multi-threaded configuration. In a single-threaded configuration publisher and subscriber execute within the main thread in a non-blocking, run-to-completion fashion. In a multi-threaded configuration, the workload is spread across a threadpool. The distribution of work is complex, but the following details are the most important from the application developer's perspective. In the publisher, data set messages and network messages are created in the threading context of the writer group although collection and transmission occur in other threads. For this reason, better performance can be achieved on a multi-core platform by distributing data items and data sets across multiple writer groups where possible rather than placing everything within one large network message. Similarly, the subscriber workload is spread across across data set readers and each data set reader executes in its own context.

Third Party Dependencies

Release 5.0 exclusively uses the WolfMQTT library as an MQTT Client for both Publisher and Subscriber. Where it is required to connect to an MQTT broker over TLS, WolfSSL must be used. Both WolfMQTT and WolfSSL are commercial software products. They are available for download from GitHub (**https://github.com/wolfSSL**) for evaluation under a GPL license, but distributing your application based on the FLEX SDK linked with any GPL code would be a serious breach of the FLEX SDK EULA. Prior to distribution of your product, commercial licenses for WolfMQTT and WolfSSL (if you require TLS) must be obtained. **When building a product for distribution, the FLEX SDK must be linked against commercially licensed versions of these libraries only**.

When building WolfMQTT, it must be built without support for MQTT version 5.0 which is not currently supported by the SDK.

TLS Client-Side Certificate Validation

Release R5.0 does not support validation of the TLS client certificate. That is, if the MQTT broker (which is the TLS server) requires the MQTT client's certificate to be validated then the SDK cannot connect to that broker. This limitation will be removed in a future release. TLS is most commonly used with server certificate validation only.

Platform Configuration

As mentioned above the FLEX SDK release 5.0 uses the WolfMQTT client library for communication with an MQTT broker. This library needs to be configured and this can be done using method calls on the IPubSubPlatformConfiguration_t interface.

The library is configured for non-blocking mode and is tightly integrated into the internal FLEX SDK threading model. If the SDK is built in a multi-threaded configuration then each MQTT connection to a broker will have its own dedicated thread for sending and receiving data, etc. The ***BrokerClientPollingIntervalInMs*** method defines the period in ms at which the library is polled. The default value is 10ms which should be suitable for most applications.

The size of the library buffers is configured by the ***TransmitBufferSizeInBytes*** and ***ReceiveBufferSizeInBytes*** methods. If the buffer sizes are less than the message sizes required to be sent and received, the library will have to be polled multiple times for each transmit and receive operation. This will reduce efficiency and may reduce performance. The default size for both of these settings is 64kB which should be suitable for most applications.

An MQTT broker will close a network connection to a connected client after a keep alive interval expires with no network activity. This time can be set with the ***DefaultBrokerKeepAliveTimeInSeconds*** method. The default value is 120 seconds. In the case of an OPC UA Publisher, key frame messages will keep the connection open but in the case of a UA Subscriber it may be necessary to periodically poll the broker in order to prevent the keep alive time from expiring. This can be configured with the ***BrokerPingIntervalInSeconds*** method. The default value is 0. For a Subscriber it is recommended to set this value to less than the keep alive time (60 seconds for example).

The network messages to be published are queued in writer group. Number of messages that can be queued is configurable in writer group using API ***NetworkMessageQueueSize*** method. Defualt value 1. After publishing a network message, next pending network message is pulled from writer group which has network message ready to be sent and published. If queue size is reached and new network message created, then oldest message will be discarded, and new message will be added to the queue. Make sure not to configure queue size to a higher number.

PubSub Configuration

A PubSub configuration defines the complete behavior of publisher and/or subscriber. The configuration defines such things as:

- What type of data is contained within a published dataset.
- Where the data is read from in the publisher address space.
- What broker the publisher and subscriber connect to.
- How the data is encoded.
- How often the data is sampled and published.
- Where in the subscriber address space the data is written to.
- Whether the publisher and/or subscriber should be enabled immediately upon creation.
- Many other behaviours.

To get a publisher up and running is a simple task when a PubSub configuration already exists. With a pre-existing configuration a publisher can be created with one function call as shown below.

```
status = PublishSubscribeFactory_t::CreatePublishSubscribe(
  IPublishSubscribe_t::CONFIGURE_PUBLISHER_ONLY,
  configuration,
  addressSpace,
  noOfThreads,
  cyclicRateInMs,
  publisher);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
UASDK_RETURN_UNEXPECTED_IF_NULL(publisher);
```

The configuration is of type PubSubConfigurationDataType_t. There are no dependencies on client or server but the PublishSubscribeFactory does require an address space to read the published data from. This address space can be stand-alone or a server address space.

In the absence of a configuration there are several ways to obtain one.

- Create a configuration programmatically.
- Load a configuration from a file (a local file or a file exposed in a server).
- Read a configuration by reading from the PubSub configuration model in a server.
- Create a configuration by writing to and making method calls on a configuration model in a server.

Creating a configuration programmatically

A PubSub configuration is defined by an instance of PubSubConfigurationDataType_t. This structure is defined in the specification at:

https://reference.opcfoundation.org/v104/Core/docs/Part14/6.2.11/

The specification is extremely granular and has many settings. In order to simplify configuration, the SDK provides a helper class that provides a degree of abstraction in manipulating the configuration structure. This helper class is called PubSubConfigurationHelper_t.

Writing a Configuration to a File

The following example code shows how to write a configuration to a file.

```
  Status_t SerialiseConfiguration(
    const IAddressSpace_t& as,
    PubSubConfigurationDataType_t& configuration)
  {
    StorageBuffer_t buffer;
    Status_t status = buffer.Initialise(1024 * 1024);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

    PubSubConfigurationHelper_t helper(configuration);

    status = helper.SerialiseConfiguration(as, buffer);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

    const char* filename = "./configuration.uab";

    IntrusivePtr_t<StdFile_t> configFile = new RefCount_t<StdFile_t>();
    UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(configFile);

    IntrusivePtr_t<String_t> filename_ = new RefCount_t<String_t>();
    UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(filename_);

    status = filename_->CopyFrom(filename);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

    status = configFile->Initialise(true, filename_);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

    uintptr_t context = 0;
    status = configFile->Open(false, true, true, false, context);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

    status = configFile->Write(context, buffer);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

    status = configFile->Close(context);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

    return OpcUa_Good;
  }
```

As you can see, we serialise the configuration into a buffer and then write the buffer into a file.

Reading a Configuration from a Local File

The following example shows how to read a configuration from a file in the local file system.

```
   Status_t DeserialiseConfiguration(
    const IAddressSpace_t& as,
    IntrusivePtr_t<PubSubConfigurationDataType_t>& configuration)
  {
    const char* filename = "./configuration.uab";

    IntrusivePtr_t<StdFile_t> configFile = new RefCount_t<StdFile_t>();
    UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(configFile);

    IntrusivePtr_t<String_t> filename_ = new RefCount_t<String_t>();
    UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(filename_);

    Status_t status = filename_->CopyFrom(filename);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

    status = configFile->Initialise(false, filename_);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

```
        ByteString_t data;
        uintptr_t context = 0;
        status = configFile->OpenForReading(context);
        UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

        status = configFile->Read(context, static_cast<int32_t>(configFile->Size()), data);
        UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

        status = configFile->Close(context);
        UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

        WrapBuffer_t buffer;
        buffer.InitialiseAsFullAndReadOnly(data);

        ArrayUA_t<String_t> destNamespaceUris;
        status = PubSubConfigurationHelper_t::DeserialiseConfiguration(as, buffer, configuration, destNamespaceUris)
;
        UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
        UASDK_RETURN_UNEXPECTED_IF_NULL(configuration);

        return OpcUa_Good;
      }
```

As you can see, we read the configuration from the file into a byte array and then deserialise this into a configuration using the helper.

When a configuration is serialized the namespace indices of all node ids and qualified names are stored per the source address space namespace array.

When the configuration is deserialized the namespace indices of all node ids and qualified names are changed to match the destination address space namespace array. The variable, destNamespaceUris, contains the namespace uris used in the configuration in the order of the address space namespaceArray (excluding the OPC UA standard namespace). There may be additional entries in this array that are not present in the address space namespaceArray. If required, the application developer can add these new entries to the namespaceArray.

Exposing a configuration File in a Server Address Space

A configuration file can be exposed as a file in a server address space. A client with appropriate permissions can then read the configuration from the server and provide it to a PubSub application. Examples of how to expose a file in a server are available from Matrikon support.

Exposing a configuration as a Model

After creating an instance of a PublishSubscriber class using PublishSubscribeFactory_t as described above, the configuration can be exposed in the address space by calling IPublishSubscribe_t:: CreateModel(). This model can then be read by a client and can be modified by a client with appropriate permissions.

Note: If there are huge number of samples to be published, try sharing accross multiple connections, groups and writers/readers.

Example configuration of a Publisher using the helper

Instantiate the helper providing the configuration you wish to create or modify.

```
PubSubConfigurationHelper_t helper(configuration);
```

If you want the publisher to be enabled by default on creation then set the relevant field.

```
configuration.Enabled() = true;
```

Create a PublishedDataSet of data items (sampled data values).

```
static const char DATASET_NAME[] = "TestDataSet";

String_t dataSetName;
Status_t status = dataSetName.CopyFrom(DATASET_NAME);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

IntrusivePtr_t<PublishedDataSetDataType_t> publishedDataSet;
status = helper.CreatePublishedDataSetOfDataItems(dataSetName, publishedDataSet);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

Add a known pre-existing node attribute to the PublishedDataSet. This adds the field to the DataSetMetaData and to the PublishedDataItems.

```
  NodeIdNumeric_t testNodeId(1, 10);
  IntrusivePtr_t<IVariableNode_t> node = addressSpace->FindVariableNode(testNodeId, status);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
  UASDK_RETURN_UNEXPECTED_IF_NULL(node);
```

```
IntrusivePtr_t<const QualifiedName_t> browseName = node->BrowseName();
UASDK_RETURN_UNEXPECTED_IF_NULL(browseName);

Status_t status = helper.AddDataItemToPublishedDataSet(
  *addressSpace,
  dataSetName,
  browseName->Name(),
  sampleValue,
  -1,
  *node->NodeId(),
  AttributeId_t::ATTRIBUTE_ID_VALUE,
  100);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

Create a connection.

```
static const char CONNECTION_NAME[] = "TestConnection";
String_t connectionName;
status = connectionName.CopyFrom(CONNECTION_NAME);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

static const char BROKER_URL[] = "mqtt://test.mosquitto.org:1883";
String_t brokerUrl;
status = brokerUrl.CopyFrom(BROKER_URL);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

BrokerConnectionTransportDataType_t connectionTransportSettings;
ArrayUA_t<KeyValuePair_t> connectionProperties;
status = connectionProperties.Initialise(0U);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

IntrusivePtr_t<PubSubConnectionDataType_t> connection;
status = helper.CreateConnectionForMqttJson(
  connectionName,
  true,
  brokerUrl,
  connectionTransportSettings,
  connection);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
UASDK_RETURN_UNEXPECTED_IF_NULL(connection);
```

Create a WriterGroup.

```
static const char GROUP_NAME[] = "TestWriterGroup";
String_t groupName;
status = groupName.CopyFrom(GROUP_NAME);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

IntrusivePtr_t<BrokerWriterGroupTransportDataType_t> groupTransportSettings
  = new SafeRefCount_t<BrokerWriterGroupTransportDataType_t>();
UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(groupTransportSettings);

groupTransportSettings->RequestedDeliveryGuarantee().Value()
  = BrokerTransportQualityOfService_t::OPCUA_BROKER_TRANSPORT_QUALITY_OF_SERVICE_ATMOST_ONCE;

status = groupTransportSettings->QueueName().CopyFrom("Mtkon/publisher/test");
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

JsonNetworkMessageContentMask_t nwMsgContentMask;
nwMsgContentMask.NetworkMessageHeader(true);
nwMsgContentMask.PublisherId(true);
nwMsgContentMask.DataSetMessageHeader(true);

IntrusivePtr_t<JsonWriterGroupMessageDataType_t> groupMessageSettings
  = new SafeRefCount_t<JsonWriterGroupMessageDataType_t>();
UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(groupMessageSettings);

groupMessageSettings->NetworkMessageContentMask() = nwMsgContentMask;
ArrayUA_t<KeyValuePair_t> groupProperties;
status = groupProperties.Initialise(0U);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

IntrusivePtr_t<WriterGroupDataType_t> writerGroup;
status = helper.CreateWriterGroup(
  *connection,
  groupName,
  true,
  MessageSecurityMode_t::OPCUA_MESSAGE_SECURITY_MODE_NONE,
  1 * 1000 * 1000,
  1000,
  5000,
  groupProperties,
  groupTransportSettings,
  groupMessageSettings,
  writerGroup);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
UASDK_RETURN_UNEXPECTED_IF_NULL(writerGroup);
```

Create a DataSetWriter.

```
  static const char WRITER_NAME[] = "TestDataSetWriter";
  String_t writerName;
  status = writerName.CopyFrom(WRITER_NAME);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  ArrayUA_t<KeyValuePair_t> dataSetWriterProperties;
  status = dataSetWriterProperties.Initialise(0U);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  IntrusivePtr_t<BrokerDataSetWriterTransportDataType_t> writerTransportSettings
    = new SafeRefCount_t<BrokerDataSetWriterTransportDataType_t>();
  UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(writerTransportSettings);

  JsonDataSetMessageContentMask_t dsMsgContentMask;
  dsMsgContentMask.DataSetWriterId(true);
  dsMsgContentMask.SequenceNumber(true);
  dsMsgContentMask.Timestamp(true);
  dsMsgContentMask.Status(true);
  dsMsgContentMask.MessageType(true);

  IntrusivePtr_t<JsonDataSetWriterMessageDataType_t> writerMessageSettings
    = new SafeRefCount_t<JsonDataSetWriterMessageDataType_t>();
  UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(writerMessageSettings);

  writerMessageSettings->DataSetMessageContentMask() = dsMsgContentMask;

  DataSetFieldContentMask_t dataSetFieldContentMask;
  dataSetFieldContentMask.StatusCode(true);
  dataSetFieldContentMask.SourceTimestamp(true);

  IntrusivePtr_t<DataSetWriterDataType_t> dataSetWriter;
  status = helper.CreateDataSetWriter(
    *writerGroup,
    writerName,
    true,
    dataSetFieldContentMask,
    10,
    dataSetName,
    dataSetWriterProperties,
    writerTransportSettings,
    writerMessageSettings,
    dataSetWriter);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
  UASDK_RETURN_UNEXPECTED_IF_NULL(dataSetWriter);
```

You now have a basic publisher configuration that you can use to create a PublishSubscribe instance using the factory.

Example configuration of a Subscriber using the helper

Instantiate the helper providing the configuration you wish to create or modify.

```
  PubSubConfigurationHelper_t helper(configuration);
```

If you want the subscriber to be enabled by default on creation then set the relevant field.

```
  configuration.Enabled() = true;
```

Add a PublishedDataSet.

```
  static const char DATASET_NAME[] = "TestDataSet";
  String_t dataSetName;
  Status_t status = dataSetName.CopyFrom(DATASET_NAME);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  IntrusivePtr_t<PublishedDataSetDataType_t> publishedDataSet;
  status = helper.CreatePublishedDataSet(dataSetName, publishedDataSet);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

Add a field to the DataSetMetaData. This defines a field in the DataSetMetaData without defining where the value is obtained from in the publisher address space.

```
    String_t fieldName;
    status = fieldName.CopyFrom("Double 10");
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

    Double_t sampleValue;

    status = helper.AddFieldToDataSetMetaData(
      *addressSpace,
      dataSetName,
      fieldName,
      sampleValue,
      -1);
    UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

Create a connection.

```
  static const char CONNECTION_NAME[] = "TestConnection";
  String_t connectionName;
  status = connectionName.CopyFrom(CONNECTION_NAME);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  static const char BROKER_URL[] = "mqtt://test.mosquitto.org:1883";
  String_t brokerUrl;
  status = brokerUrl.CopyFrom(BROKER_URL);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  BrokerConnectionTransportDataType_t connectionTransportSettings;

  IntrusivePtr_t<PubSubConnectionDataType_t> connection;
  status = helper.CreateConnectionForMqttJson(
    connectionName,
    true,
    brokerUrl,
    connectionTransportSettings,
    connection);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
  UASDK_RETURN_UNEXPECTED_IF_NULL(connection);
```

Add a ReaderGroup.

```
IntrusivePtr_t<ReaderGroupDataType_t> readerGroup;

String_t readerGroupName;
status = readerGroupName.CopyFrom("TestReaderGroup");
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

ArrayUA_t<KeyValuePair_t> groupProperties;
status = groupProperties.Initialise(0U);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

status = helper.CreateReaderGroup(
  *connection,
  readerGroupName,
  true,
  MessageSecurityMode_t::OPCUA_MESSAGE_SECURITY_MODE_NONE,
  64 * 1024,
  groupProperties,
  0,
  0,
  readerGroup);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
UASDK_RETURN_UNEXPECTED_IF_NULL(readerGroup);
```

Add a DataSetReader.

```
String_t dataSetReaderName;
status = dataSetReaderName.CopyFrom("TestDataSetReader");
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

ArrayUA_t<KeyValuePair_t> dataSetReaderProperties;
status = dataSetReaderProperties.Initialise(0U);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

//Transport settings
IntrusivePtr_t<BrokerDataSetReaderTransportDataType_t> readerTransportSettings
  = new SafeRefCount_t< BrokerDataSetReaderTransportDataType_t>();
UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(readerTransportSettings);

readerTransportSettings->RequestedDeliveryGuarantee().Value()
  = BrokerTransportQualityOfService_t::OPCUA_BROKER_TRANSPORT_QUALITY_OF_SERVICE_ATMOST_ONCE;
```

```
status = readerTransportSettings->QueueName().CopyFrom("Mtkon/publisher/test");
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

//NetworkMessageContentMask
JsonNetworkMessageContentMask_t nwMsgContentMask;
nwMsgContentMask.NetworkMessageHeader(true);
nwMsgContentMask.DataSetMessageHeader(true);

//DataSetMessageContentMask
JsonDataSetMessageContentMask_t dsMsgContentMask;
dsMsgContentMask.DataSetWriterId(true);
dsMsgContentMask.SequenceNumber(true);
dsMsgContentMask.Timestamp(true);
dsMsgContentMask.Status(true);
dsMsgContentMask.MessageType(true);

//message settings
IntrusivePtr_t<JsonDataSetReaderMessageDataType_t> readerMessageSettings
  = new SafeRefCount_t<JsonDataSetReaderMessageDataType_t>();
UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(readerMessageSettings);

readerMessageSettings->NetworkMessageContentMask() = nwMsgContentMask;
readerMessageSettings->DataSetMessageContentMask() = dsMsgContentMask;

DataSetFieldContentMask_t dataSetFieldContentMask;
dataSetFieldContentMask.StatusCode(true);
dataSetFieldContentMask.SourceTimestamp(true);

IntrusivePtr_t<DataSetReaderDataType_t> dataSetReader;

status = helper.CreateDataSetReader(
  *readerGroup,
  dataSetReaderName,
  true,
  0,
  0,
  0,
  publishedDataSet->DataSetMetaData(),
  dataSetFieldContentMask,
  60000,
  10,
  MessageSecurityMode_t::OPCUA_MESSAGE_SECURITY_MODE_NONE,
  dataSetReaderProperties,
  readerTransportSettings,
  readerMessageSettings,
  0,
  dataSetReader);
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
UASDK_RETURN_UNEXPECTED_IF_NULL(readerGroup);
```

Create a SubscribedDataSet. This defines where the field is dispatched to in the subscriber address space.

```
  IntrusivePtr_t<TargetVariablesDataType_t> targetVariables;
  status = helper.CreateSubscribedDataSetOfTargetVariables(*dataSetReader, targetVariables);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
  UASDK_RETURN_UNEXPECTED_IF_NULL(targetVariables);

  UASDK_RETURN_UNEXPECTED_IF_FALSE(publishedDataSet->DataSetMetaData().Fields().Count() == 2);
  UASDK_RETURN_UNEXPECTED_IF_FALSE(publishedDataSet->DataSetMetaData().Fields()[0].is_set());
  UASDK_RETURN_UNEXPECTED_IF_FALSE(publishedDataSet->DataSetMetaData().Fields()[1].is_set());

  NumericRange_t emptyRange;

    NodeIdNumeric_t targetId(1, 100001);

  status = helper.AddTargetVariableToSubscribedDataSet(
    *dataSetReader,
    *addressSpace,
    publishedDataSet->DataSetMetaData().Fields()[0]->DataSetFieldId(),
    emptyRange,
    targetId,
    AttributeId_t::ATTRIBUTE_ID_VALUE,
    emptyRange,
    OverrideValueHandling_t::OPCUA_OVERRIDE_VALUE_HANDLING_LAST_USABLE_VALUE,
    0);
  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

You now have a basic subscriber configuration that you can use to create a PublishSubscribe instance using the factory.

Creating Event Filters

When configuring a publisher to publish event notifications, event filters can be defined. These event filters determine what events and what properties or attributes of the events should be published to the dataset.

An EventFilter contains select clauses and a where clause. The select clauses define what properies or attributes of the events are published and the where clause defines what events are published.

The SDK provides helper APIs to create the event filters and these methods can be found in the EventHelper_t class.

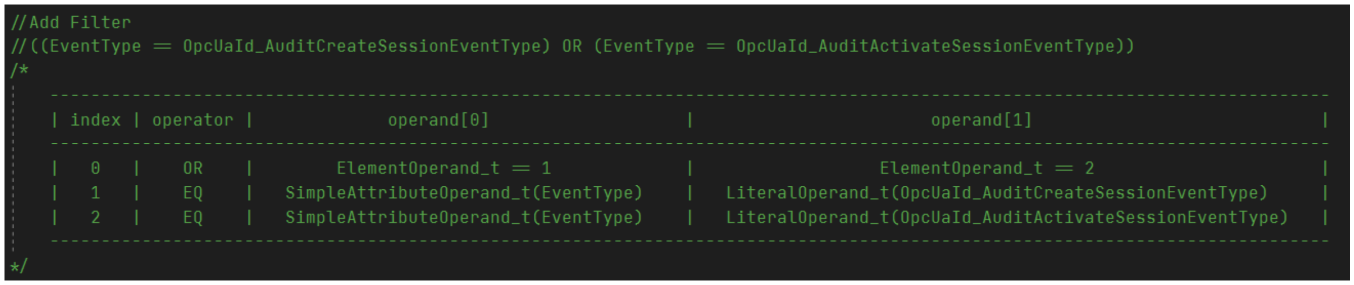
Method to define select clause/selected attributes/properites:

- AddSimpleAttributeOperand()

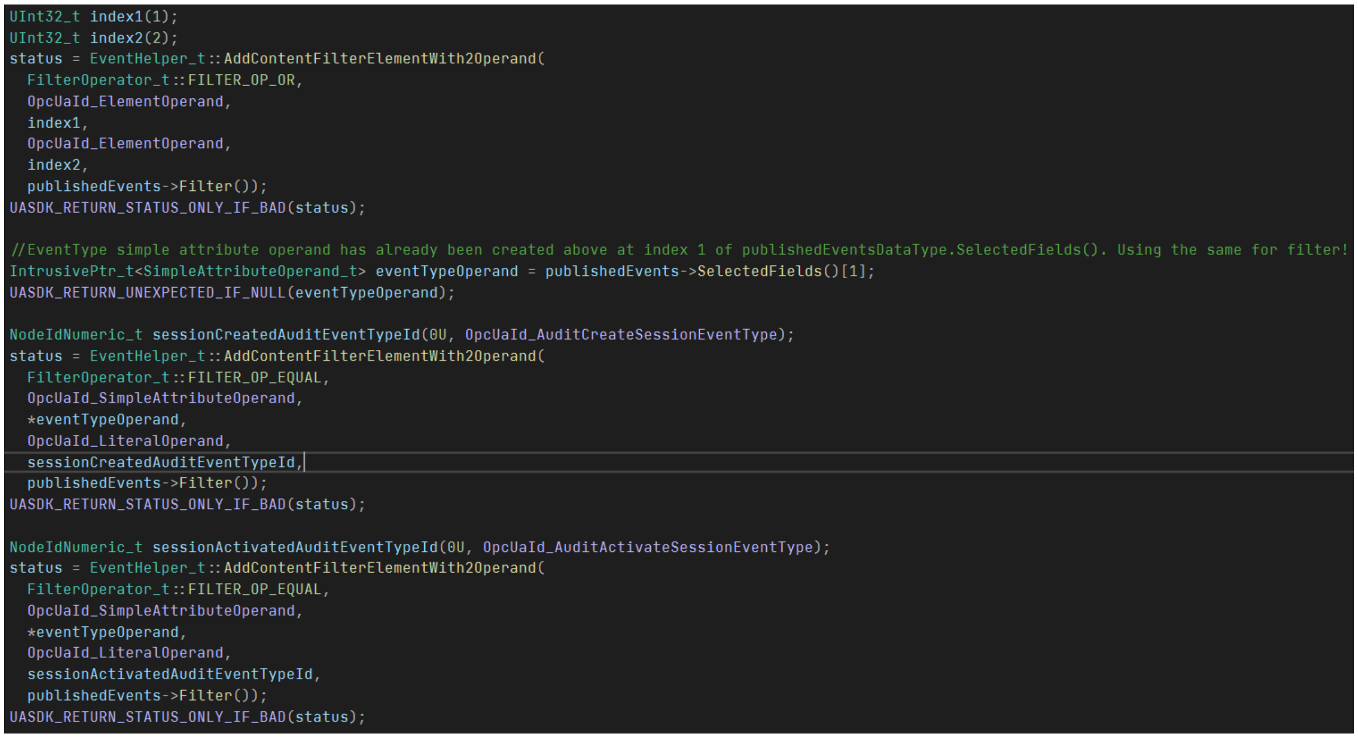
Methods to define where clause:

- AddContentFilterElementWith1Operand()
- AddContentFilterElementWith2Operand()
- AddContentFilterElementWith3Operand()

Here is an example of the creation of three event filters, as defined in the table below.



```
//Add Filter
//((EventType == OpcUaId_AuditCreateSessionEventType) OR (EventType == OpcUaId_AuditActivateSessionEventType))
/*
    -----------------------------------------------------------------------------------------------------------------------------
    | index | operator |             operand[0]                |                         operand[1]                         |
    -----------------------------------------------------------------------------------------------------------------------------
    |   0   |    OR    |        ElementOperand_t == 1          |                    ElementOperand_t == 2                   |
    |   1   |    EQ    |   SimpleAttributeOperand_t(EventType) |   LiteralOperand_t(OpcUaId_AuditCreateSessionEventType)    |
    |   2   |    EQ    |   SimpleAttributeOperand_t(EventType) |   LiteralOperand_t(OpcUaId_AuditActivateSessionEventType)  |
    -----------------------------------------------------------------------------------------------------------------------------
*/
```



```
UInt32_t index1(1);
UInt32_t index2(2);
status = EventHelper_t::AddContentFilterElementWith2Operand(
  FilterOperator_t::FILTER_OP_OR,
  OpcUaId_ElementOperand,
  index1,
  OpcUaId_ElementOperand,
  index2,
  publishedEvents->Filter());
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

//EventType simple attribute operand has already been created above at index 1 of publishedEventsDataType.SelectedFields(). Using the same for filter!
IntrusivePtr_t<SimpleAttributeOperand_t> eventTypeOperand = publishedEvents->SelectedFields()[1];
UASDK_RETURN_UNEXPECTED_IF_NULL(eventTypeOperand);

NodeIdNumeric_t sessionCreatedAuditEventTypeId(0U, OpcUaId_AuditCreateSessionEventType);
status = EventHelper_t::AddContentFilterElementWith2Operand(
  FilterOperator_t::FILTER_OP_EQUAL,
  OpcUaId_SimpleAttributeOperand,
  *eventTypeOperand,
  OpcUaId_LiteralOperand,
  sessionCreatedAuditEventTypeId,
  publishedEvents->Filter());
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

NodeIdNumeric_t sessionActivatedAuditEventTypeId(0U, OpcUaId_AuditActivateSessionEventType);
status = EventHelper_t::AddContentFilterElementWith2Operand(
  FilterOperator_t::FILTER_OP_EQUAL,
  OpcUaId_SimpleAttributeOperand,
  *eventTypeOperand,
  OpcUaId_LiteralOperand,
  sessionActivatedAuditEventTypeId,
  publishedEvents->Filter());
UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

API References

Refer to the HTML version of API reference (API_Reference.html) in the SDK Document folder.

Security APIs

Security configuration is an optionally called simple application.

RSA application specifies the following:

- The key length for basic128rsa15 and Basic256 is between 1024 bits to 2048bits. The default value is 1024bits.
- The key length for basic256sha256 policy is between 2048bits to 4096bits. The default value is 2048bits.

Note: Key length 1024 bits is not considered safe to use anymore. Please make sure that even for RSA minimum 2048 key length has been used.

To help the application developer with the security, SDK has implemented the interface classes which can be used for validation of certificate, getting the certificate, adding the certificate to the stack etc. Developer can make use of APIs defined in these classes based on the need. Interface classes are:

- ICertificateStore_t
- ISecurityPolicy_t
- IX509Certificate_t

SDK has also provided a static class **Crypto_t** to create certificate store objects (concrete object of ICertificateStore_t). This class can be used to symmetric and Asymmetric encryption, decryption and verification as well.

For each group of certificate (Application Group, User Group and/or HTTPS Group), developer can create the certificate store. These certificate stores (concrete object of ICertificateStore_t) will hold the own certificates and private key, trusted and issuer certificates and CRLs. To create Application stores developer can use the API called **Crypto_t::CreateCertificateStore()**. For providing the password for private key and either to store, temporary trsut or permanently trust the remote certificate, developer has to implement the concrete class of **ICertificateStoreCallback_t** and pass that object to **Crypto_t::CreateCertificateStore()** API. Following code snippets describes on configuring the application with security.

Create certificate store

Below snippet will provide the example of creating the application group certificate store.

```
String_t pkiPath;

Status_t status = pkiPath.CopyFrom("./");

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);


//Validation Option for certificate store

CertificateValidationOptions_t validationOptions;


IntrusivePtr_t<CertificateStoreCallback_t> certStoreCallback = new
SafeRefCount_t<CertificateStoreCallback_t>();

UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(certStoreCallback);


//Application Certificate Store

IntrusivePtr_t<ICertificateStore_t> appCertStore;

status = Crypto_t::CreateCertificateStore(pkiPath,
UA_CERTIFICATE_GROUP_DEFAULT_APPLICATION_GROUP, certStoreCallback, appCertStore);
```

```
UASDK_RETURN_BAD_STATUS_IF_NULL(appCertStore, status);


appCertStore->SetDefaultCertificateValidationOptions(validationOptions);


status = appCertStore->RefreshCache();

if (status.IsBad())

{

  //Ignore Initial Error.

  //Log the Error if required!

  //UASDK_LOG_STATUS_ONLY_IF_BAD(status);

}
```

For storing the certificate in the directory, the path can be provided with the help of *pkiPath* variable. Variable *certStoreCallback* is the concrete class of **ICertificateStoreCallback_t**. Based on the requirement *validationOptions* can be updated. API **RefreshCache()** is very important, this API has to be called to update the certificate store object (concrete object of ICertificateStore_t) with the certificates, private key, trusted and issuer stacks (certificates and CRLs). Any change in the pki path (directory where the certificates are stored), to cache those changes, developer has to call **RefreshCache()** API.

Validating the Certificate

Below snippet will provide the example of validating the RSA 256 certificate; also if validation fails, creating the RSA 256 certificate.

```
//Create Generation parameter created if required.

CertificateGenerationParameters_t parameters;

UASDK_RETURN_UNEXPECTED_IF_NULL(configuration->ApplicationURI());

status = parameters.applicationUri.CopyFrom(*configuration->ApplicationURI());

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

bool refreshAndValidationRequired = false;
```

```
String_t hostname;

status = hostname.CopyFrom("HostName1");

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

//RSA 256 Cert

status = appCertStore->ValidateOwnCertificate( UA_CERTIFICATE_TYPE_RSA_SHA_256_APPLICATION_TYPE,

  validationOptions, hostname, *configuration->ApplicationURI(), false, true);

if(status.IsBad())

{

  status = InitialiseCertGenParam(UA_CERTIFICATE_TYPE_RSA_SHA_256_APPLICATION_TYPE, parameters);

  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  status = certStoreCallback->GetPrivateKeyFilePassword(

    UA_CERTIFICATE_GROUP_DEFAULT_APPLICATION_GROUP,

    UA_CERTIFICATE_TYPE_RSA_SHA_256_APPLICATION_TYPE,

    parameters.privateKeyPassword);

  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  IntrusivePtr_t<IX509Certificate_t> rsa256Cert;

  status = appCertStore->CreateSelfSignedCertificate(parameters, rsa256Cert);

  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  refreshAndValidationRequired = true;

}
```

In case of validation of the exisiting certificate in the *pkiPath* fails, the developer will need to create new certificate for that certificate type (RSA Min or RSA 256). Here '*parameters'* is the variable of type **CertificateGenerationParameters_t**, the developer will need to update this variable with the correct values to create the valid certificate and private key. The variable *refreshAndValidationRequired* is used to refresh again after creating the certificate to make sure the certificate created is valid.

Before validating the created certificate(s) developer has to refresh the certificate store cache using the API **RefreshCache()**. Below snippet will demonstrates the same.

```
if (refreshAndValidationRequired)

{

  status = appCertStore->RefreshCache();

  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  status = appCertStore->ValidateOwnCertificate(

    UA_CERTIFICATE_TYPE_RSA_MIN_APPLICATION_TYPE, validationOptions,

    hostname, *configuration->ApplicationURI(), false, true);

  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

  status = appCertStore->ValidateOwnCertificate(

    UA_CERTIFICATE_TYPE_RSA_SHA_256_APPLICATION_TYPE, validationOptions,

    hostname, *configuration->ApplicationURI(), false, true);

  UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

}
```

Now the certificate store is ready for use by application. This store must be configured into the application using the following APIs:

- AppCertificateStore() for Application group certificates (Client and Server)
- UserCertificateStore() for User group certificates (Server Only)

Code snippet for this is:

```
configuration->AppCertificateStore(appCertStore);
```

variable *configuration* is object of concrete type of interface IServerConfiguration_t (in case of server) and IClientConfiguration_t (in case of client).

Crypto Library APIs

Irrespective of the Crypto library used, the application developer must ensure to define or implement the below mentioned functions. Examples for OpenSSL, MbedTLS and NanoSSL are provided as a part of SDK. The Application developer can refer to the crypto library to understand the functions and how to implement the same for the required security binding.

Different security APIs in which specific crypto library calls can be made are mentioned below:

UASecurity_library_initialise()

This API is used to initialize the security crypto library and random number generator method. The API is as follows:

UASecurity_library_initialise(const UA_UTF8_string_t* pki_path)

This API initializes the security crypto library using pki_path. This path is selected as default and is not a mandatory field. It returns OpcUa_Good in case of successful initialization; else it returns a specific error code.

UASecurity_library_terminate()

This API is used to de-initialize the security crypto library and the security related platform components. The API is as follows:

UASecurity_library_terminate(void)

This function returns void.

UASecurity_rand_bytes()

This API is used to generate random numbers. The API is as follows:

UASecurity_rand_bytes(UA_Byte_string_t* buf)

This API generates random numbers stored in a buffer. This buffer has pre-allocated memory to hold the required bytes of random numbers. The required number of bytes is notified in **max_length** field of the buffer. It returns OpcUa_Good in the case of successful random number generation; else it returns a specific error code.

UASecurity_rand_pseudo_bytes()

This API is used to generate random numbers. The API is as follows:

UASecurity_rand_pseudo_bytes(UA_Byte_string_t* buf)

The API generates the random numbers and the numbers are stored in the buffer. This buffer has pre-allocated memory to hold the required bytes of random numbers. The required number of bytes is notified in **max_length** field of the buffer. It returns OpcUa_Good in the case of successful random number generation, else returns a specific error code.

UASecurity_symmetric_encrypt()

This function is used for the symmetric encryption of the plain text. The API is as follows:

UA_Status_t UASecurity_symmetric_encrypt(

UA_Sym_Cipher_t type, const UA_Byte_string_t* iv,

const UA_Byte_string_t* key, const UA_Byte_string_t* plaintext,

UA_Byte_string_t* ciphertext)

This API is used to encrypt plain text using specified symmetric *type* parameter and produce an encrypted text. The type parameter indicates the type of symmetric method used for encrypting the plain text. Parameter '*iv*' describes the initialization vector for symmetric encryption and it is a random number. The *key* parameter is the AES key input for plain text encryption. Parameter *plaintext* is the input message for encryption. '*cipher-text*' is the encrypted output message. It returns OpcUa_Good in the case of successful symmetric encryption; else it returns a specific error code.

UASecurity_symmetric_decrypt()

This API is used for the symmetric decryption of the encrypted texts. The API is as follows:

UASecurity_symmetric_decrypt(

UA_Sym_Cipher_t type, const UA_Byte_string_t* iv,

const UA_Byte_string_t* key, const UA_Byte_string_t* ciphertext,

UA_Byte_string_t* plaintext)

The API decrypts cipher text and produces plain text as output. The *type* parameter indicates the type of symmetric method used to encrypt the plain text. Parameter '*iv*' describes the initialization vector for symmetric decryption and is a random number. The *key* parameter is the AES key input for decryption. *Ciphertext* parameter indicates the cipher text input for symmetric decryption. It returns OpcUa_Good in the case of successful symmetric decryption; else it returns a specific error code.

UASecurity_digest()

This API is used for message digest. The API is as follows:

UASecurity_digest(

UA_Digest_t type, const UA_Byte_string_t* data,

UA_Byte_string_t* hash)

This API is used to perform hash operation on the input data. The *type* parameter indicates the type of digest method used to hash the input data. *Data* parameter indicates the actual data to be hashed. The output *hash* parameter indicates the hashed value. It returns OpcUa_Good in the case of successful message digest, else it returns a specific error code.

UASecurity_hmac()

This API is used to perform hash on the input text using HMAC (Keyed-Hash Message authentication code). The API is as follows:

UASecurity_hmac(

UA_Digest_t type, const UA_Byte_string_t* key,

const UA_Byte_string_t* data, UA_Byte_string_t* hash)

The *type* parameter indicates the type of digest method used to hash the input text. The *key* parameter specifies the HMAC key required to hash. HMAC gives authentication to the message. *Data* parameter indicates the actual message data. Output *hash* parameter indicates the hashed value. It returns OpcUa_Good in case of successful hash; else it returns a specific error code.

UASecurity_p_sha()

This API is used to hash the input text using P-SHA method. The API is as follows:

UASecurity_p_sha(

UA_Digest_t type, const UA_Byte_string_t* secret,

const UA_Byte_string_t* seed, UA_Byte_string_t* output)

The *type* parameter indicates the type of digest method used to hash the input text. The *secret* parameter is the secret key for performing P-SHA operation. *Seed* is the nonce for P-SHA and it is a random number which is shared during handshaking process with the client. *Output* parameter indicates the output P-SHA value. It returns OpcUa_Good in case of successful hash; else it returns a specific error code.

UASecurity_asymmetric_encrypt()

This API is used for asymmetric encryption of plain text based on the cipher type and certificate. The API is as follows:

UASecurity_asymmetric_encrypt(

UA_Asym_Cipher_t type, const UA_Byte_string_t* receiver_chain,

//handle to the certificate const UA_Byte_string_t* plaintext,

UA_Byte_string_t* ciphertext)

The *type* parameter indicates the asymmetric cipher type. Parameter *receiver_chain* holds the certificate byte string in der format. *Plaintext* is the input message for encryption. Parameter *cipher text* is the output encrypted message. It returns OpcUa_Good in case of successful asymmetric encryption; else it returns a specific error code.

UASecurity_asymmetric_decrypt()

This API is used for asymmetric decryption of cipher text based on Asymmetric cipher type and private key. The API is as follows:

UASecurity_asymmetric_decrypt(

UA_Asym_Cipher_t type, uintptr_t receiver_private_key_handle,

const UA_Byte_string_t* ciphertext, UA_Byte_string_t* plaintext)

The *type* parameter indicates the asymmetric cipher type. Parameter *receiver_private_key_handle* indicates the private key to handle decryption of input message. *Handle* is an internal library format. This format varies based on the Include library. Parameter *cipher text* is the input message for decryption. *Plaintext* is an output decrypted message. It returns OpcUa_Good in the case of successful asymmetric decryption; else it returns a specific error code.

UASecurity_asymmetric_sign()

This API is used to sign the message. The message could be any byte string. The API is as follows:

UASecurity_asymmetric_sign(

UA_Digest_t type, SecurityPolicy_t policy, uintptr_t signee_private_key_handle, const UA_Byte_string_t* data, UA_Byte_string_t* signature)

The parameter *type* indicates the type of digest algorithms used to hash the message. Paramter policy is the security policy used for connection. Parameter *signee_private_key_handle* is the private key handle to sign the message. *Data* indicates the actual input text message of the sender. Parameter *signature* contains the actual signature text. It returns OpcUa_Good in case of successful signature of the message, else it returns a specific error code.

UASecurity_asymmetric_verify()

This API is used to verify the message signature. The API is as follows:

UASecurity_asymmetric_verify(

UA_Digest_t type, SecurityPolicy_t policy, const UA_Byte_string_t* sender_chain, const UA_Byte_string_t* data, const UA_Byte_string_t* signature)

The parameter *type* indicates the type of digest algorithms used for hashing the message. Paramter policy is the security policy used for connection. Parameter *sender_chain* is certificate byte string in der format. *Data* indicates the actual input text message of the sender. Parameter *signature* contains the actual signature text. It returns OpcUa_Good in the case of successful verification of the message signature, else it returns a specific error code.

UASecurity_certificate_validate()

This API is used to validate the certificates. The API is as follows:

UASecurity_certificate_validate(

const UA_Certificate_Validation_Parameters_t* parameters,

bool_t validate_trust, int32_t* rejected_certificate_length)

The input *parameters* contain complete or partial certificate chain required for validation. *Validate_trust* indicates if the certificate requires trust validation or not. Parameter *rejected_certificate_length* indicates the rejected certificate length in byte string in the case of validation failure on reasons of trust. It returns OpcUa_Good in the case of successful validation of the certificates, else it returns a specific error code.

UASecurity_certificate_validation_parameters_init()

This API is used to initialize the validation parameter structure. The API is as follows:

UASecurity_certificate_validation_parameters_init(

UA_Certificate_Validation_Parameters_t* parameters)

This API updates the structure with a default value. It returns OpcUa_Good in the case of successful initialization of the validation parameter structure, else it returns a specific error code.

UASecurity_get_certificate_key_size()

This API is used to get the key size of a certificate. The API is as follows:

UASecurity_get_certificate_key_size(

uintptr_t certificate_handle, uint32_t* key_size_in_bytes)

Parameter *certificate_handle* indicates the handle to the certificate. Return parameter *key_size_in_bytes* give the actual key size of a certificate. It returns OpcUa_Good in the case of successful actual size return of a certificate, else it returns a specific error code.

UASecurity_get_certificate_max_plaintext_size()

This API is used to get the maximum size of the plain text which is used for encryption. The API is as follows:

UASecurity_get_certificate_max_plaintext_size(

UA_Asym_Cipher_t type, uintptr_t certificate_handle,

uint32_t* max_plaintext_size)

Parameter *type* specifies asymmetric cipher type. Parameter *certificate_handle* indicates the handle to the certificate. Output parameter *max_plaintext_size* returns the maximum size of the plain text. It returns OpcUa_Good in the case of successful actual size return of the plain text, else it returns a specific error code.

UASecurity_der_to_certificate()

This API is used for converting a certificate from der to library specific format (handle). The API is as follows:

UASecurity_der_to_certificate(

UA_Byte_string_t* certificate_der, uintptr_t* certificate_handle)

Handle is an internal library format. This format varies based on include library. The interpreter is used to make it portable. Parameter *certificate_der* indicates the certificate data in der format. Parameter *certificate_handle* gives the output in handle format. It returns OpcUa_Good in the case of successful output in handle format, else it returns a specific error code.

UASecurity_der_to_crl()

This API is used to convert the crl in der format to specific forman (hanlde). The API is as follows:

UASecurity_der_to_crl(

UA_Byte_string_t* crl_der, uintptr_t* crl_handle)

Handle is an internal library format. This format varies based on include library. The interpreter is used to make it portable. Parameter *crl_der* indicates the CRL data in der format. Parameter *crl_handle* gives the output in handle format. It returns OpcUa_Good in the case of successful output in handle format, else it returns a specific error code.

UASecurity_encrypt_pem_private_key()

This API is used to encrypt the private key in PEM format. The API is as follows:

UASecurity_encrypt_pem_private_key(

const UA_Byte_string_t* pkcs8_pem_private_key,

const UA_UTF8_string_t* private_key_password,

UA_Byte_string_t* pkcs8_encrypted_pem_private_key)

Parameter *pkcs8_pem_private_key* indicates the private key in pem with pkcs8 standard. Parameter *private_key_password* password to encrypt the private key. Paramter *pkcs8_encrypted_pem_private_key* is the private key which has been encryted with pkcs8 standard using the password provided. It returns OpcUa_Good in the case of successful output, else it returns a specific error code.

UASecurity_get_private_key_handle()

This API is used to get the private key in handle format. The API is as follows:

UASecurity_get_private_key_handle(

const UA_Byte_string_t* pkcs8_pem_private_key, UA_UTF8_string_t*

private_key_password, uintptr_t* private_key_handle)

Parameter *pkcs8_pem_private_key* specifies the PKCS8 PEM encoded private key. Parameter *private_key_password* indicates the password for the private key. Parameter *private_key_handle* specifies handle for the private key. It returns OpcUa_Good in the case of successful out of private key for handle format, else it returns a specific error code.

UASecurity_get_new_certificate_stack_handle()

This API is used to get the new stack handle for certificate. The API is as follows:

UASecurity_get_new_certificate_stack_handle(uintptr_t* stack_handle)

Output parameter *stack_handle* will hold the stack handle for certificate. It returns OpcUa_Good in the case of successful out of private key for handle format, else it returns a specific error code.

UASecurity_get_new_crl_stack_handle()

This API is used to get the new stack handle for CRL. The API is as follows:

UASecurity_get_new_crl_stack_handle(uintptr_t* stack_handle)

Output parameter *stack_handle* will hold the stack handle for CRL. It returns OpcUa_Good in the case of successful out of private key for handle format, else it returns a specific error code.

UASecurity_add_certificate_to_stack()

This API is used to add the certificate to the stack. The API is as follows:

UASecurity_add_certificate_to_stack(

uintptr_t certificate_handle, uintptr_t stack_handle)

Parameter *certificate_handle* will have the certificate handle information which need to be added to the stack. Paramter *stack_handle* is the stach in which certificate handle must be added. It returns OpcUa_Good in the case of successful out of private key for handle format, else it returns a specific error code

UASecurity_add_crl_to_stack()

This API is used to add the CRL to the stack. The API is as follows:

UASecurity_add_crl_to_stack(

uintptr_t crl_handle, uintptr_t stack_handle)

Parameter *crl_handle* will have the CRL handle information which need to be added to the stack. Paramter *stack_handle* is the stach in which CRL handle must be added. It returns OpcUa_Good in the case of successful out of private key for handle format, else it returns a specific error code.

UASecurity_get_leaf_certificate_der()

This API is used to get the leaf certificate in chain of certificates. The API is as follows:

UASecurity_get_leaf_certificate_der(

UA_Byte_string_t* certificate_chain_der, UA_Byte_string_t* leaf_certificate_der)

Parameter *certificate_chain_der* will have the chain of certificate in der. Output paramter *leaf_certificate_der* will hold th leaf certificate. It returns OpcUa_Good in the case of successful out of private key for handle format, else it returns a specific error code

Note: This API will not allocate any memory.

UASecurity_free_certificate_handle()

This API is used to free the allocated memory blocks for a certificate in handle format. The API is as follows:

UASecurity_free_certificate_handle(

uintptr_t certificate_handle)

Certificate_handle parameter specifies the pointer of the certificates in handle format. It returns OpcUa_Good in the case of successful output, else it returns a specific error code.

UASecurity_free_crl_handle()

This API is used to free the allocated memory blocks for a CRL in handle format. The API is as follows:

UASecurity_free_crl_handle(uintptr_t crl_handle)

crl_handle parameter specifies the pointer of the CRL in handle format. It returns OpcUa_Good in the case of successful output, else it returns a specific error code.

UASecurity_free_private_key_handle()

This API is used to free the allocated memory blocks for the private key handle. The API is as follows:

UASecurity_free_private_key_handle(

uintptr_t private_key_handle)

private_key_handle parameter specifies handle pointer to the private key. It returns OpcUa_Good in the case of successful output, else it returns a specific error code.

UASecurity_free_certificate_stack_handle()

This API is used to free the allocated memory blocks for the certificate stack handle. The API is as follows:

UASecurity_free_certificate_stack_handle(

uintptr_t certificate_stack_handle)

Certificate_stack_handle parameter specifies handle pointer to the certificate stack handle. It returns OpcUa_Good in the case of successful output, else it returns a specific error code.

UASecurity_free_crl_stack_handle()

This API is used to free the allocated memory blocks for the crl stack handle. The API is as follows:

UASecurity_free_crl_stack_handle(

uintptr_t crl_stack_handle)

crl_stack_handle parameter specifies handle pointer to the CRL stack handle. It returns OpcUa_Good in the case of successful output, else it returns a specific error code.

UASecurity_create_self_signed_certificate()

This API is used to generate a self-signed certificate. The API is as follows:

UASecurity_create_self_signed_certificate(

const UA_Certificate_Generation_Parameters_t* parameters, UA_Byte_string_t* certificate_der, UA_Byte_string_t* private_key_pem)

Input *parameters* are the parameter required for certificate generation. The output parameter *certificate_der* will hold the newly created self signed certificate in der format and output parameter *private_key_pem* will hold the private key in pem format. It returns OpcUa_Good in the case of successful generation of self-signed certificate, else it returns a specific error code.

UASecurity_certificate_generation_parameters_init()

This API is used to initialize the self-sign parameters. The API is as follows:

UASecurity_certificate_generation_parameters_init(

UA_Certificate_Generation_Parameters_t* parameters)

Input *parameters* are the parameter required for certificate generation. Parameters are initialized with a default value. It returns OpcUa_Good in the case of successful initialization of self-sign parameters, else it returns a specific error code.

UASecurity_create_certificate_signing_request()

This API is used to create certificate signing request. The API is as follows:

UASecurity_create_certificate_signing_request(

UA_Certificate_Generation_Parameters_t* parameters, bool_t regeneratePrivateKey, uintptr_t current_certificate_handle, uintptr_t current_private_key_handle, UA_Byte_string_t* certificate_signing_request_der, UA_Byte_string_t certificate_signing_request_der)**

Input *parameter* indicates the parameters like Subject Name, Issuer Name, URI, Domain Name etc. Parameter *regeneratePrivateKey* is a Boolean variable which indicates whether to generate a private key or not. Paramter *current_certificate_handle is the handle to the certificate.* Paramter current_private_key_handle is the handle to the private key, Output parameter *certificate_signing_request_der* gives the certificate signing request in der format. Output *private_key_pem* represents the private key in PEM format only if the *regeneratePrivateKey* has ben set to TRUE. It returns OpcUa_Good in the case of successful creation of signing request, else it returns a specific error code.

UASecurity_get_certificate_issued_date()

This API is used to get the certificate issued date. The API is as follows:

UASecurity_get_certificate_issued_date(

uintptr_t certificate_handle, int64_t * issued_time)

This API function gets the certificate issued date of the input certificate. Parameter *certificate_handle* is the certificate handle from which issued date to be extracted. Output parameter issued_time will hold the issued time infomration. It returns OpcUa_Good in the case of successful creation of the directory, else it returns a specific error code.

UASecurity_get_certificate_expiry_date()

This API is used to get the certificate expiry date. The API is as follows:

UASecurity_get_certificate_expiry_date(

const UA_Byte_string_t* certificate_handle, int64_t * expiry_time)

This API function gets the certificate expiry date of the input certificate. Parameter *certificate_handle* is the expiry date of the input certificate. Output parameter expiry_time will hold the expiry time infomration. It returns OpcUa_Good in the case of successful creation of the directory, else it returns a specific error code

UASecurity_crl_is_issuer()

This API is used to check if the CRL provided is the issuer CRL or not. The API is as follows:

UASecurity_crl_is_issuer (uintptr_t crl_handle,

uintptr_t certificate_handle, bool_t* result)

Paramter *crl_handle* is handle fo the CRL. Parameter *certificate_handle* is the handle to the certificate. Output parameter *result* will be set to TRUE is the CRL is issuer CRL. It returns OpcUa_Good in the case of successful initialization of self-sign parameters, else it returns a specific error code.

UASecurity_get_certificate_uri()

This API is used to check if the CRL provided is the issuer CRL or not. The API is as follows:

UASecurity_get_certificate_uri(

uintptr_t certificate_handle, UA_Byte_string_t* uri)

Parameter *certificate_handle* is the handle to the certificate. Output parameter *uri* will be set to with the URI in the certificate. It returns OpcUa_Good in the case of successful initialization of self-sign parameters, else it returns a specific error code.

UASecurity_get_certificate_hostnames()

This API is used to get the certificate hostnames. The API is as follows:

UASecurity_get_certificate_uri(

uintptr_t certificate_handle, UA_Byte_string_t hostnames[UAX509_SAN_MAX_HOSTNAMES], uint32_t* no_of_results)

Parameter *certificate_handle* is the handle to the certificate. Output parameter *hostnames* will be set to with the hostnames present in the certificate. Output parameter *no_of_results* will have the number of hostnames which are prenet in the output hostname argument. It returns OpcUa_Good in the case of successful initialization of self-sign parameters, else it returns a specific error code.

UASecurity_get_certificate_ip_addresses()

This API is used to get the ip addresses present in the certificate. The API is as follows:

UASecurity_get_certificate_uri(

uintptr_t certificate_handle, UA_Byte_string_t ip_addresses[UAX509_SAN_MAX_HOSTNAMES], uint32_t* no_of_results)

Parameter *certificate_handle* is the handle to the certificate. Output parameter *ip_addresses* will be set to with the ip addresses present in the certificate. Output parameter *no_of_results* will have the number of ip addresses which are prenet in the output ip_addresses argument. It returns OpcUa_Good in the case of successful initialization of self-sign parameters, else it returns a specific error code.

UASecurity_free_bytestring_data()

This API is used to free the memory allocated by the security binding. The API is as follows:

UASecurity_free_bytestring_data(UA_Byte_string_t* bytestring)

Parameter *bytestring* is the pointer to memory location which needs to be freed. It returns OpcUa_Good in the case of successful initialization of self-sign parameters, else it returns a specific error code.

Certificate Management Helpers

Certificate management functions comprise the management and distribution of certificates and Trust Lists for OPC UA Applications. An application that provides the certificate management functions is called CertificateManager. GDS and CertificateManager will typically be combined in one application.

There are two primary models for Certificate management: pull and push management. In pull management, the application acts as a Client and uses the Methods on the CertificateManager to request and update Certificates and Trust Lists. The application is responsible for ensuring the Certificates and Trust Lists are kept up to date. In push management, the application acts as a Server and exposes Methods which the CertificateManager can call to update the Certificates and Trust Lists as required.

SDK has implemented helpers for easily achieve the push and pull management behaviours. Developer of client or server application can make use of the following classes for the same.

- CertMgmtPullHelper_t
- CertMgmtPushHelper_t

There are APIs available in each those classes which can be used to achieve the push and pull functionality with ease. Developer should create respective object and use necessary API can be used to achieve the desired functionality.

CertMgmtPushHelper_t

The CertMgmtPushHelper_t class has been created to make it easy for the application developer to support certificate push management in a UA Server. Push management is described in Part 12 of the OPC UA specification and provides a standard technique for remote provisioning and management of OPC UA Server application security. Application certificates and trust lists can be managed remotely by a suitable authorized UA Client when connected over an authenticated and encrypted secure channel.

There are three steps required to use the push management helper.

1. Load the push management information model into the server address space. This can be easily done via an XML nodeset file.
2. Implement the push helper notifier interface, ICertMgmtPushHelperNotifier_t so that your application can interact with the push helper when required.
3. Initialise the push helper via the CertMgmtPushHelper_t::Initialise() method.

When initializing the helper there are several decisions that need to be made:

1. What certificate groups do you want to manage via push. The SDK currently supports the application group and the user group.
2. What application certificate types do you want to support.
3. Do you want to require the apply changes method to be called after performing an update or do you want updates to be applied as they are made.

Once the push helper is initialized it operates in conjunction with the push helper notifier implemented by the application.

CertMgmtPullHelper_t

CertMgmtPullHelper_t has been created to help the application developer to achieve the pull management.

The pull management functionalities are as shown below:

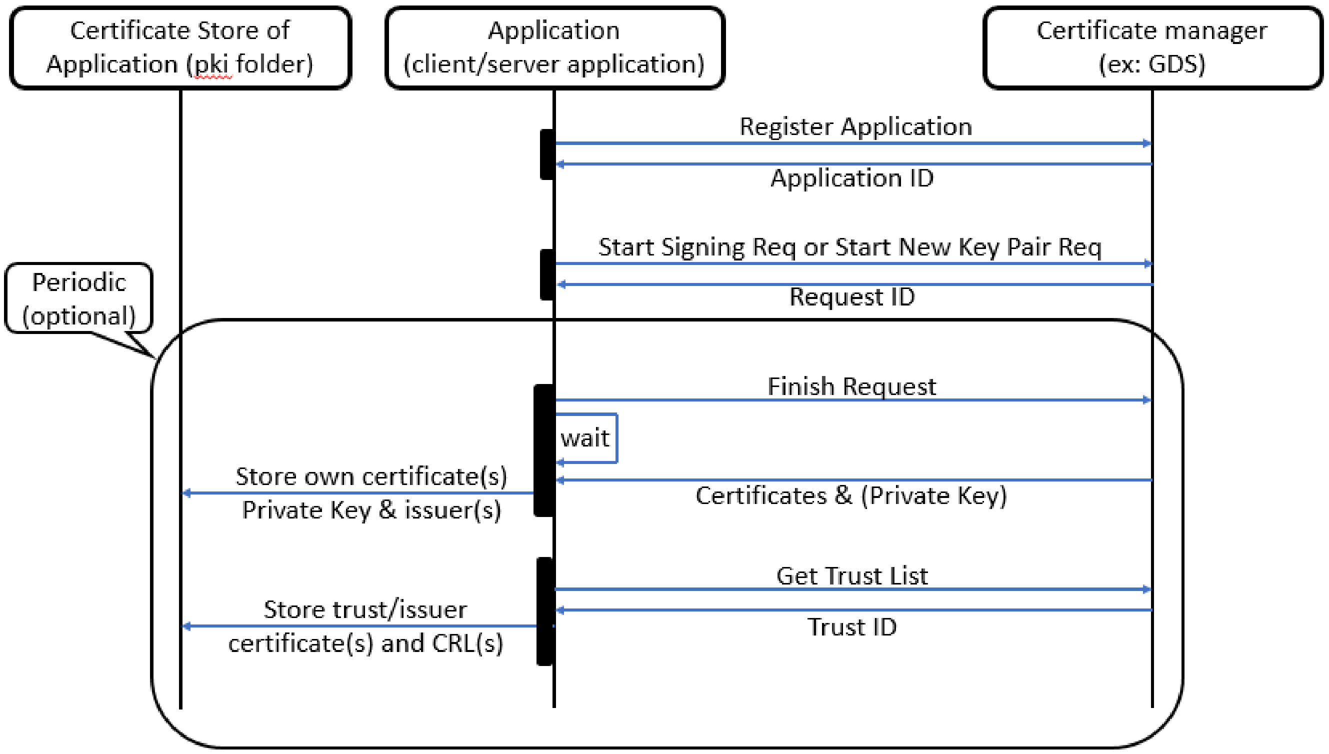

Certificate Store of Application (pki folder)
Application (client/server application)
Certificate manager (ex: GDS)
Register Application
Application ID
Start Signing Req or Start New Key Pair Req
Request ID
Periodic (optional)
Finish Request
wait
Store own certificate(s) Private Key & issuer(s)
Certificates & (Private Key)
Get Trust List
Store trust/issuer certificate(s) and CRL(s)
Trust ID

Figure 6 – Certificate Management Pull model

As it can be seen from the Figure 6 that, application will register itself with Certificate manager and call methods on the certificate manager for signing request, new key pair request, trust list etc. These things can be done periodically when certificates are expired and/or need to update the trust or issuer list.

CertMgmtPullHelper_t has implemented simple APIs to achieve the same and provided the interface **ICertMgmtPullHelperNotifier_t** to get the notification on completion of any request. Helper has provided both synchronous and asynchronous APIs. Developer can use any based on the requirement. For the all the service request sent to certificate manager can have the common timeout. **ServiceCallTimeoutInSeconds()** API can be used to set the same. If it not set, SDK will use the default value.

Intialising CertMgmtPullHelper_t

Developer can call BeginInitialise () method to initialse the CertMgmtPullHelper_t. Developer should wait for Helper to give notigfication in ICertMgmtPullHelperNotifier_t object once the initialization is completed.

In case of Client application developer can create CertMgmtPullHelper_t object and initialse the helper with the following:

- List of IApplicaiton_t object
- Client object

- Session created with GDS server
- Certificate store which should be used to update the certificate(s), private key and/or trust list
- Instance of ICertMgmtPullHelperNotifier_t object

Incase of server application, developer can either create client object inside a server and initialize with the above parameters or provide the necessary details to the helper to establish connection with GDS and get initialized. The required set of paramters are:

- List of IApplicaiton_t object
- Certificate store which should be used to update the certificate(s), private key and/or trust list
- Instance of ICertMgmtPullHelperNotifier_t object
- Instance of IServerConfiguration_t object
- Discovery server url for GDS server.

Registering application

To register an application, value of structure ApplicationRecordDataTypeParameters_t is required. Please make sure to fill the correct values for structure, as this information will be used to create the certificate by certificate manager. Application Id field in the parameter can be left empty, as this information will be provided by the Certificate manager, after successful registration.

After registration, developer can get the Application Id anypoint during the lifetime of the helper with the help of API **ApplicationId()**.

Start Signing Request

Helper has provided the API BeginPerformSigningRequest() which will call StartSigningRequest and FinishRequest() methods on certificate manager. If the certificate recived is valid, it will be stored in the pki path along with the private key and issuer certificate. Parameter regeneratePrivateKey in BeginPerformSigningRequest() will specify whether privake key has to be generated while creating the Signing Request using security binding. Parameter refreshCertStoreCache in BeginPerformSigningRequest() specifies whether certificate store object need to cache the certificates after the validation or it just has to be stored in the PKI path.

Start New Key Pair Request

Helper has provided the API BeginPerformNewKeyPairRequest() which will call StartNewKeyPairRequest() and FinishRequest() on certificate manager. If the certificate recived is valid, it will be stored in the pki path along with the private key and issuer certificate. Parameter refreshCertStoreCache in BeginPerformNewKeyPairRequest() specifies whether certificate store object need to cache the certificates after the validation or it just has to be stored in the PKI path.

As we believe Private Key should not be shared to any application for potention security threat, this feature will be OFF by default. Requesting private key from certificate manager is ideally known to certificate manager. So, this feature will be be OFF by default. Build macro **UASDK_INCLUDE_PRIVATE_KEY_REQUEST_FROM_GDS** should be set to non-zero to enable this feature.

Trust List

Helper has provided the API GetTrustList()/BeginGetTrustList() to get the trust list from certificate manager. This will call GetTrustList() on Certificate manager. Parameter refreshCertStoreCache in API specifies whether certificate store object need to cache the certificates after the validation or it just has to be stored in the PKI path.

As trust list may get changed, this API either has to be called periodically or developer can use EnableGetTrustListPeriodically() provided by the helper to achieve the same.

Certificate Status

When own certificates and/or Private keys have been received from Certificate manager, certificate might have been expired and it needs updated again. For the Helper provides API GetCertificateStatus()/BeginGetCertificateStatus() to get the certificate status. This will call GetCertificateStatus() on Certificate manager. If update required, certificate manager responds with update required as the output argument to GetCertificateStatus() method call. Developer can then update certificate either by calling BeginPerformSigningRequest() or BeginPerformNewKeyPairRequest().

Status of the certificate should be checked periodically by developer or use the API EnableGetCertificateStatusPeriodically() to achieve the same.

Build Macros

The macros that are set to 0 or 1 to achieve the required OPC UA application development are mentioned below.

Generic Build Macros

UASDK_MAX_ALIGNMENT

This macro is used for the maximum alignment of the CPU. The default value is 8.

UASDK_BIG_ENDIAN

The default value is 0. Set to 1 for big endian architectures.

UASDK_CROSS_ENDIAN_DOUBLE

Some older hardware floating point units on little endian architectures use an unusual floating-point format which is a hybrid of little and big endian. The default value is 0. Set to 1 to enable.

UASDK_NO_THROW

While implemented in C++, the SDK does not use Run Time Type Information or Exceptions. Typically, the SDK would be compiled with these options explicitly disabled in the compiler configuration. Some compilers require the no-throw exception specification to be used in several places in the SDK even when exceptions are disabled. Other compilers fail to compile if exception specifications are specified when exceptions are disabled. It is recommended not to alter without consulting with **support@matrikonopc.com** if the value needs to be changed. Set this macro to throw() or an empty macro to suit the user's compiler. The default value is throw().

Warning:

⚠ On some platforms setting this macro to anything other than throw() can result in segment violations or hard faults to be in the out of memory conditions. Hence, modification of this macro must be done with precaution.

UASDK_INCLUDE_SUBSCRIPTIONS

This macro includes support for subscriptions and monitored items. The default value is 1 (enables the feature).

UASDK_INCLUDE_COMPLEX_DATA

This macro includes support for Complex Data. The default value is 0 (disables the feature)

UASDK_INCLUDE_SECURITY

This macro enables full security functionality in the SDK. The default value is 0. Set to 1 for support.

UASDK_USE_OPENSSL

This macro enables OpenSSL PKI component interface module. The default value is 0. Set to 1 for support.

UASDK_USE_MBEDTLS

This macro enables MBedTLS PKI component interface module. The default value is 0. Set to 1 for support.

UASDK_OPTIMISE_FOR_SPEED

This macro configures the SDK to favor speed over the RAM usage. Various optimizations are affected which reduce the time to perform the Browse and TranslateBrowsePathsToNodeIds services. Message encoding and decoding time is also dramatically reduced. This macro can significantly increase memory utilization in the server which is configured with a maximum message size which is much larger than the maximum message chunk size. In such an instance, the default value is 0. Set to 1 for speed.

UASDK_USE_SYSTEM_HEAP

This macro is used to make use of a system heap instead of the internal allocator. Set the value to 0 for internal allocator and 1 for system heap. The default value is 1.

UASDK_OVERLOAD_NEW_AND_DELETE

This macro is used to overload new and delete the catch classes which are not inherited from the Allocatable_t.

UASDK_USE_FILE_OBJECT_HELPERS

This macro is used to enable the file object helper.

UASDK_UNUSED

This macro is used to remove the warnings if any argument passed to a function is not used inside the functions.

UASDK_DEADBAND_DATA_TYPE

This macro is used for dead band filter. The default value is set to double. Set to float for Cortex-M4F which only has a single precision FPU.

UASDK_INCLUDE_SERVER_LWIP_RAW

This macro is used to include LWIP TCPIP Library for HDK.

UASDK_INCLUDE_LIBRARY_BARE_METAL

This macro is used to include bare metal Library for HDK.

UASDK_INCLUDE_TCPIP_LWIP_RAW

This macro is used to include LWIP TCPIP for HDK.

UASDK_INCLUDE_RESIZABLE_ARRAYS

This macro is used to include Resizable arrays.

UASDK_THROTTLE_INBOUND_MESSAGES

This macro is used to throttle incoming messages to reduce memory usage.

UASDK_USE_CONTIGUOUS_BUFFERS

This macro is used to use contiguous buffers to increase speed.

UASDK_USE_LARGE_CONTAINERS

This macro is used to use large containers to increase speed

UASDK_SUBSCRIPTION_PURGE_NOTIFICATIONS

This macro is used to purge notifications on monitored item deletion to save memory at the

cost of speed.

UASDK_INCLUDE_SERVER_REGISTRATION_AGENT

This macro is used to include types for Discover Server Registration.

UASDK_INCLUDE_ADDITIONAL_BROWSE_NAMES

This macro is used to include Additional Browse Name String liternals

UASDK_INCLUDE_TEST_FEATURES_IN_SERVER

This macro is used to include testing support

UASDK_MIN_SERVER_CYCLIC_RATE_IN_MS

This macro ensures minimum server cyclic rate to 50ms. It limits ServerCyclicRateInMs() in server configuration. Modifying this value for a platform without submiliseconds timing precision may lead to unexpected behaviour.

UASDK_MIN_CLIENT_CYCLIC_RATE_IN_MS

This macro ensures minimum client cyclic rate to 50ms. It limits ClientCyclicRateInMs() in client configuration. Modifying this value for a platform without submiliseconds timing precision may lead to unexpected behaviour.

UASDK_INCLUDE_ADDRESS_SPACE_LISTENER_EXTENDED

This Macro is used to use the extened feature of address space listener.

UASDK_INCLUDE_LTTNG_TRACE

This macro is used to use LTTng for tracing (Linux only)

Services specific

Below are some of the macros which are required for specific services.

UASDK_INCLUDE_METHOD_SERVICE_SET

This macro is used to include types for method service set. Exclude to reduce code size.

UASDK_INCLUDE_NODE_MANAGEMENT_SERVICE_SET

This macro is used to include types for node management service set. Exclude to reduce code size.

UASDK_INCLUDE_TRANSFER_SUBSCRIPTIONS_SERVICE

This macro is used to include types for transfer subscriptions service. Exclude to reduce code size.

UASDK_INCLUDE_QUERY_SERVICE_SET

This macro is used to include types for query service set. Exclude to reduce code size.

UASDK_INCLUDE_ADDITIONAL_DISCOVERY_SERVICES

This macro is used to include types for additional discovery services. Exclude to reduce code size.

UASDK_INCLUDE_ADDITIONAL_DA_TYPES

This macro is used to include additional DA types. Exclude to reduce code size.

UASDK_INCLUDE_METHOD_SERVICE_SET

This macro must include methods when subscriptions are selected.

Data Access, Events and History

Below are some of the macros which are required for Data Access, Events and History

UASDK_USE_HELPERS

This macro is used to enable the helper modules for DA or file object.

UASDK_USE_DATA_ACCESS_HELPERS

This macro is used to include the DA related nodes. Set the value to 0 to exclude the data access related nodes and 1 to include the data access related nodes.

UASDK_INCLUDE_EVENTS

This macro includes support for Events. The default value is 0 (disables the feature).

UASDK_INCLUDE_ALARMS_AND_CONDITIONS

This macro includes support for Alarms and conditions. The default value is 0 (disables the feature)

UASDK_INCLUDE_HISTORY_ACCESS

This macro includes History Access support. Defualt value is 0 (disables the feature)

UASDK_INCLUDE_AGGREGATES

This macro includes aggregates support. Default value is 0 (disables the feature)

UASDK_USE_HISTORY_ACCESS_HELPERS

This macro includes history access helpers support. Default value is 1.

Certificate Management

Below are the macros which are required for certificate management helpers.

UASDK_INCLUDE_CERTIFICATE_MANAGEMENT_PULL_HELPER

This macro includes support for certificate management pull helper. Default value is 0.

UASDK_INCLUDE_CERTIFICATE_MANAGEMENT_PUSH_HELPER

This macro includes support for certificate management push helper. Default value is 0.

UASDK_INCLUDE_PRIVATE_KEY_REQUEST_FROM_GDS

This macro includes support for private key request from GDS. Default value is 0.

Logging and Debug

Below are some of the macros which are required for logging or debugging.

UASDK_ASSERTION

This macro enables internal SDK assertions.

UASDK_DEBUG

This macro enables the debugging interface of the SDK and log generic warnings and errors.

UASDK_DEBUG_SHOW_FUNCTION_NAME

This macro logs the function name when outputting debug log messages. This increases the SDK binary size.

UASDK_DEBUG_SHOW_FILE_AND_FUNCTION_NAME

This macro logs the file and function names when outputting debug log messages. This increases the SDK binary size.

UASDK_DEBUG_SESSIONS

This macro logs session state changes.

UASDK_DEBUG_CHANNELS

This macro logs secure channel state changes.

UASDK_DEBUG_CONNECTIONS

This macro logs UA TCP connection state changes.

UASDK_DEBUG_SUBSCRIPTIONS

This macro logs subscription activity.

UASDK_DEBUG_MESSAGES

This macro logs messages sent between components of the SDK.

UASDK_DEBUG_MEMORY

This macro logs memory allocations and deallocations when using the internal allocator.

Server Diagnostics

Macros required to enable the server diagnostics.

UASDK_INCLUDE_SERVER_DIAGNOSTICS_SUMMARY

This macro is used to include server diagnostics summary.

UASDK_INCLUDE_SUBSCRIPTION_DIAGNOSTICS

This macro is used to include subscription diagnostics.

UASDK_INCLUDE_SESSION_DIAGNOSTICS

This macro is used to include session diagnostics and session security diagnostics.

Miscellaneous

This section, mentions how to use some of the features of the SDK.

Registering Server to Discovery Server

Ensure that the following Build Macros are set to 1 before registering the server.

- **UASDK_INCLUDE_SERVER_REGISTRATION_AGENT**
- **UASDK_INCLUDE_CLIENT**
- **UASDK_INCLUDE_SERVER**

RegistrationAgent_t

RegistrationAgent_t module is used to register the server to a Local Discovery Server. Follow the steps below to register a server to the Local Discovery Server.

1. Create an object of RegistrationAgent_t.
2. Initialize RegistrationAgent_t with discovery server's URL.
3. Call RegisterServer2() with periodic registration time interval.

Note:

1. Ensure to store the Server's certificate in the trusted certificates folder of Local discovery server.
2. If RegisterServer2() service gets BadNotSupported as response, RegistrationAgent_t will call RegisterServer(). The discovery server returns BadNotSpported as response if the RegisterServer2() service has not been implemented in it.

Locales and Translations

The following functions requires to be implemented by the application developer for localization support.

- **GetLocalizedText(const ArrayUA_t < String_t > & locales, Status_t & result)**
  - This API returns the localized text value with the given locale information.
- **GetLocalizedText(Status_t & result)**
  - This API returns the default localized text value.

Complex Type Support

Introduction

The Matrikon FLEX OPC UA SDK supports application defined complex data types. Application defined complex data types are types that are defined by the application at runtime in the server address space. Instances of complex data types can then be used as the value attributes of variables nodes in the server. The SDK provides classes that facilitate usage of complex data types including encoding and decoding of instances.

FLEX SDK complex type components

Several classes are provided by the SDK to enable the application developer work with application defined complex data types and these are introduced below.

Data types

Part 3 of the OPC UA specification (version 1.04) defines a new optional attribute of data type nodes in the server address space. The attribute is a DataTypeDefinition which provides the information required to define the components of a data type and its encodings. There are two concrete subtypes of this attribute, EnumDefinition and StructureDefinition. The EnumDefinition defines the values and names for application defined enumerations. The StructureDefinition defines application defined complex types, or structures along with their fields.

The following are two subtypes of the DataTypeDefinition attribute:

- EnumDefinition: This describes the values and names for application-defined enumerations.
- StructureDefinition: This describes the application-defined complex types or structures along with their fields.

Application defined structures can be normal structures, structures with optional fields or unions. They can inherit from a hierarchy of abstract and/or concrete structures each with their own fields. They can contain fields of any concrete type including other structure types.

In addition to data type classes that map directly to the attributes defined in the standard, the SDK also contains following extended classes that add additional functionality:

- StructureDefinitionComplete_t
- StructureFieldComplete_t

These classes contain complete data type definitions for all nested fields of a given structure allowing for high performance encoding/decoding of an instance with no dynamic address space or type cache lookups.

ILocalizableDataTypeDefinition_t

DataTypeDefinition attributes can be added to the server address space automatically via XML nodeset import or via the IDataTypeNode_t API. Data type definitions can optionally contain localized elements and, to allow the application developer to support localization of these elements (if required), the ILocalizableDataTypeDefinition_t interface is provided to allow the application developer control dynamic generation or lookup of localized data type definitions.

UnknownStructure_t

When creating an instance of an application defined complex type the class used is UnknownStructure_t. As the name indicates, the type is unknown to the SDK unlike a known complex type such as Range_t for example. An unknown structure instance contains an array of fields that can be populated with values in accordance with how the structure is defined. For example, a union would have an array of fields but only one of them would be populated with a value. A structure with optional fields may have only some of its fields populated, whereas a normal structure would be expected to contain values for all define fields.

An unknown structure instance also contains a pointer to its definition (an instance of StructureDefinitionComplete_t). Many instances can share the same definition. When encoding an unknown structure, it requires a pointer to its definition so that it knows how to encode itself.

Decoding is a two-step process. The FLEX SDK protocol stack will create an unknown structure when it encounters a structure type that it does not recognize. It cannot decode the structure so instead it stores the encoded byte stream inside the instance along with the encodingId. The application developer on receipt of the instance can add the definition if he already has it (or look it up if he does not) and then complete the decoding.

When creating an unknown structure instance that contains another structure as a field there are many options for implementing the field instance inside the unknown structure. If the structure is created by a decoder, it will use unknown structures for all structure fields (both known and unknown). If the structure is created by the application it can use either unknown structures or known structures (where available) to model the field instance.

When creating an unknown structure instance that contains a simple data type as a field there are several options for implementing the field instance inside the unknown structure. If the structure is created by a decoder it will use built in types for all simple type fields. If the structure is created by the application it can use either built in types or simple types (where available) to model the field instance.

The application should however be consistent in the approach as comparisons between structures using different classes for structure fields will always return a negative result.

UADataTypeInfo_t

This class is initialized with a pointer to an address space and optionally a list of preferred locales. On the server side the actual server address space is used whereas on the client side a type cache is used which exposes the same interface as the server address space. The application developer can use the class to obtain either raw or complete type information for any data type that is present in the cache or address space. When requesting complete type information, the class will automatically traverse the cache so as to create a complete definition for the type and all of its dependencies.

ClientDataTypeHelper_t

On the client side a type cache is required to be created, to use UADataTypeInfo_t for type lookups. The client API creates the cache for the application developer but it will initially be empty. The ClientDataHelper_t class will interact with a remote server address space over a pre-existing session and execute browse and read service calls to populate the type cache with the information needed for a given data type.

Working with the FLEX SDK complex type classes

Defining an application defined structure

On the server side the first step to using application defined complex types is to expose the type definition in the server address space. This defines how to encode and decode the type. There are two ways to do this,

- via an XML nodeset import
- programmatically via the API.

Below is shown a simple type definition in XML.

```
<UADataType NodeId="ns=1;s=MyValveCalibrationType" BrowseName="MyValveCalibrationType">

  <DisplayName>MyValveCalibrationType</DisplayName>

  <References>

    <Reference ReferenceType="HasSubtype" IsForward="false">i=22</Reference>

  </References>

  <Definition Name="MyValveCalibrationType">

    <Field Name="Gain" DataType="i=11" />

    <Field Name="Sensitivity" DataType="i=8" IsOptional="true" />

  </Definition>

</UADataType>



<UADataType NodeId="ns=1;s=MyValveType" BrowseName="MyValveType">

  <DisplayName>MyValveType</DisplayName>

  <References>

    <Reference ReferenceType="HasSubtype" IsForward="false">i=22</Reference>

    <Reference ReferenceType="HasEncoding" IsForward="true">ns=1;s=MyValveType Binary
Encoding</Reference>

  </References>
  <Definition Name="MyValveType">

    <Field Name="Open Time" DataType="i=294" />

    <Field Name="Close Time" DataType="i=294" />

    <Field Name="Calibration" DataType="ns=1;s=MyValveCalibrationType" />

  </Definition>

</UADataType>



<UAObject NodeId="ns=1;s=MyValveType Binary Encoding" BrowseName="Default Binary"
SymbolicName="DefaultBinary">

  <DisplayName>Default Binary</DisplayName>

  <References>

    <Reference ReferenceType="HasEncoding" IsForward="false">ns=1;s=MyValveType</Reference>

    <Reference ReferenceType="HasTypeDefinition">i=76</Reference>

  </References>

</UAObject>
```

The node ids are in string format for convenience in the example but they can be of any format. Numeric node ids typically result in better performance than string node ids. We have defined a new data type, MyValveType, that represents a valve. It inherits from the OPC Foundation Structure type and has a binary encoding node showing that the server can encode/decode the data type in UA Binary. The structure has three fields, two of which are of type UtcTime and one of which is of type MyValveCalibrationType. MyValveCalibrationType has also been defined and as can be seen it has no encoding. This means that it cannot be encoded or decoded in an extension object but only as a field of another structure.

Creating a server-side instance of an application defined structure

On the server side to create a functional instance of the MyValveType we must perform the following:

1. Using the data type info class traverse the address space extracting a complete type definition for the new type.

```
//Create and intialise the typeInfo tool

Status_t status;
```

```
UADataTypeInfo_t typeInfo;

ArrayUA_t<String_t> locales;

status = typeInfo.Initialise(addressSpace, locales);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

//Create the node id of the data type

NodeIdString_t myValveTypeId;

myValveTypeId.NamespaceIndex() = 1;

myValveTypeId.Identifer().CopyFrom("MyValveType");

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

//Retrieve the complete data type definition of the data type

IntrusivePtr_t<const StructureDefinitionComplete_t> myValveDefinition;

status = typeInfo.CreateStructureDefinitionCompleteFromTypeId(myValveTypeId,
myValveDefinition);

UASDK_RETURN_BAD_STATUS_IF_NULL(myValveDefinition, status);
```

2. Create an instance of an unknown structure and make it represent an instance of MyValveType by associating it with the type definition.

```
//Create an instance of the type and associate it with the type definition

IntrusivePtr_t<UnknownStructure_t> myValve = new SafeRefCount_t<UnknownStructure_t>();

UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(myValve);

myValve->Definition() = myValveDefinition;
```

3. Populate the fields of the instance. There are many ways of doing so and these are shown below.

```
//Using the SetField() method without validation

IntrusivePtr_t<UtcTime_t> openTime = new SafeRefCount_t<UtcTime_t>();

UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(openTime);

String_t openTimeFieldName;
```

```
status = openTimeFieldName.CopyFrom("OpenTime");

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

status = myValve->SetField(openTimeFieldName, openTime);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

//Assigning to the field directly

IntrusivePtr_t<UtcTime_t> closeTime = new SafeRefCount_t<UtcTime_t>();

UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(closeTime);

uint32_t noOfExpectedFields = 3;

if ((myValve->Fields().Count() != myValveDefinition->Fields().Count())

  && (myValve->Fields().Count() != noOfExpectedFields))

{

  UASDK_RETURN_BAD_STATUS(OpcUa_BadUnexpectedError);

}

uint32_t fieldIndexForCloseTime = 1;

myValve->Fields()[1] = closeTime;

//Using the SetField() method (for a nested field) with validation

IntrusivePtr_t<Double_t> gain = new SafeRefCount_t<Double_t>();

UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(gain);

Array_t<String_t> gainFieldName;

status = Utilities_t::PopulateArrayOfStrings("Calibration/Gain", '/', gainFieldName);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

status = myValve->SetField(gainFieldName, gain, true);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

4. Validate the data contained within the fields of the instance to ensure that it follows the type definition. If this is not done a client reading the instance may result in encoding errors.

```
//Validate the instance prior to using it as a value attribute. This will verify that the
data

//corresponds to the data type definition

status = myValve->Validate();

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

Client-side reading of an application defined structure as the value attribute of a variable

1. Create a type cache. The type cache is created by the SDK client class and has the same interface as the server address space. This cache can be used to store type information for all types that the client interacts with.

```
//Create an empty client side type cache

Status_t status;

IntrusivePtr_t<IClientConfiguration_t> configuration = uaclient->GetConfiguration(status);

UASDK_RETURN_BAD_STATUS_IF_NULL(configuration, status);

IntrusivePtr_t<IAddressSpace_t> typeCache = configuration->CreateEmptyTypeCache(1000,
status);

UASDK_RETURN_BAD_STATUS_IF_NULL(typeCache, status);

ClientDataTypeHelper_t typeHelper;

status = typeHelper.Initialise(typeCache);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

2. Populate the type cache with the type information for the MyValveType. In this example, we do this before reading the instance of the type but this can also be done as a reaction to receiving a value of a type which is not recognized.

```
//Populate the type cache with the type information for MyValveType

IntrusivePtr_t<ClientTypeLookupCompleted_t> callback = new
SafeRefCount_t<ClientTypeLookupCompleted_t>();

NodeIdString_t typeId;

typeId.NamespaceIndex() = 1;

typeId.Identifer().CopyFrom("MyValveType");
```

```
status = typeHelper.BeginLookupTypeInformationFromTypeId(session, typeId, 0, callback);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

while (!callback->completed)

{

  uaclient->Poll();

}

if (callback->result.IsBad())

{

  UASDK_RETURN_BAD_STATUS(callback->result.Value());

}
```

3. Read an instance of the type from the value attribute of a variable node in the server.

```
//Read an instance of MyValveType from the value attribute of a variable node in the server

IntrusivePtr_t<ReadRequest_t> readRequest = new SafeRefCount_t<ReadRequest_t>();

UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(readRequest);

readRequest->Header().TimeoutHint().Value(1000);

IntrusivePtr_t<ReadValueId_t> rvi = new SafeRefCount_t<ReadValueId_t>();

UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(rvi);

NodeIdString_t variableId;

variableId.NamespaceIndex() = 1;

status = variableId.Identifer().CopyFrom("My Variable");

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

status = variableId.CopyToNodeId(rvi->NodeId());

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

rvi->AttributeId().Value(AttributeId_t::ATTRIBUTE_ID_VALUE);
```

```
status = readRequest->NodesToRead().Initialise(1);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

readRequest->NodesToRead()[0] = rvi;

IntrusivePtr_t<ReadResponse_t> readResponse;

status = session->Read(readRequest, readResponse);

if (status.IsBad())

{

  UASDK_RETURN_BAD_STATUS(status.Value());

}

IntrusivePtr_t<UnknownStructure_t> myValve = RuntimeCast<UnknownStructure_t*>(*readResponse-
>Results()[0]->Value());
```

4. Create the complete type definition of the received value from the type cache and decode the value once the instance has been associated with the type definition.

```
//Create the complete type definition of the received value from its encodingId and decode
the value

UADataTypeInfo_t typeInfo;

ArrayUA_t<String_t> locales;

status = typeInfo.Initialise(typeCache, locales);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

IntrusivePtr_t<const StructureDefinitionComplete_t> definition;

status = typeInfo.CreateStructureDefinitionCompleteFromEncodingId(*myValve-
>BinaryEncodingId(), definition);

UASDK_RETURN_BAD_STATUS_IF_NULL(definition, status);

myValve->Definition() = definition;

status = myValve->Decode();

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

5. Read a field from the instance. In this example, we use the field name to look up the field value and write the value to stdout.

```
//Read a field using a field name

String_t fieldName;

status = fieldName.CopyFrom("OpenTime");

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

IntrusivePtr_t<BaseDataType_t> field = myValve->GetField(fieldName);

if (field.is_set())

{

  IntrusivePtr_t<String_t> valueAsString = field->ToUtf8String();

  UASDK_RETURN_OUT_OF_MEMORY_IF_NULL(valueAsString);

  ScopedPtr_t<Array_t<char> > resourceHolder;

  const char* cstr = valueAsString->ToCString(resourceHolder);

  if (cstr != 0)

  {

    printf("Field value is %s\n", cstr);

  }

}
```

Client-side persistence of the type cache

Rather than rebuilding the type cache every time a client connects to a server the client can optionally persist the cache to a file or other data store and reload it when the client application restarts as follows.

```
//Persist the type cache to a data store

IntrusiveList_t<const INode_t> allNodesInCache;

status = typeCache->GetAllNodesAsAList(allNodesInCache);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

```
uint32_t maxBufferSizeInBytes = 1024 * 1024;

StorageBuffer_t buffer;

status = buffer.Initialise(maxBufferSizeInBytes);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

status = typeCache->ExportNodesToByteArray(allNodesInCache, true, buffer);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

//Now contents of buffer can be persisted to a data store
```

The following are the restrictions if executing the above step:

- A given type cache must only be used for a specific server. If connecting subsequently to another server the type information in the cache will likely be incorrect and/or incomplete.
- DataTypeDefinitions contain localizable data. The ClientDataTypeHelper will populate the type cache with data in the locale specified when the session was activated (assuming the server has that locale available). If different locales are required a new type cache must be created and populated with type information obtained from a different session.

Converting between type representations

The SDK contains many classes representing known types and many of these are structures. An example would be the Argument structure which is defined in the class Argument_t. When the SDK protocol stack receives an extension object over the wire it attempts to decode the payload using a known type class. This means that, if a class exists in the SDK for a particular structure, then an instance of that class will be created during the decoding process. Only if no class exists for the encodingId will the SDK create an instance of unknown structure for subsequent decoding by the application (once it has looked up the type information and associated it with the unknown structure instance).

In some cases, on the client side, it may be convenient to always interact with unknown structures as this makes it easy to display the values and definitions of fields while using a very small code base. Unknown structures only ever contain other unknown structures and built in types making them very easy to work with. In this case any structure can be converted to an unknown structure as follows.

```
Argument_t argument;

UnknownStructure_t unknownStructure;

status = unknownStructure.CopyFrom(argument);

UASDK_RETURN_STATUS_ONLY_IF_BAD(status);

unknownStructure.Definition() = argumentDefinition;
```

```
        status = unknownStructure.Decode();

        UASDK_RETURN_STATUS_ONLY_IF_BAD(status);
```

The conversion process implemented in the method UnknownStructure_t::CopyFrom(const Structure_t&) performs the conversion by encoding the input into a byte string. The maximum length of the encoded data can be optionally specified (the default max length is 1024 bytes).

Reverse Connect

For connection protocols such as TCP, the ReverseHello Message allows Servers behind firewalls with no open ports to connect to a Client and request that the Client establish a SecureChannel using the socket created by the Server.

For message based protocols the *ReverseHello Message* allows *Servers* to announce their presence to a *Client*. In this scenario, the *EndpointUrl* specifies the *Server's* protocol specific address and any tokens required to access it.

If application developed is Server, as server has to announce its presence to the client, Server application should call **InitiateReverseConnection()** on UAServere_t object with client endpoint. This will announce the server to client. To get the status of the connection after server announces itself, developer can implement the class inheriting from **IConnectionListener_t** interface and take necessary actions in the implmenetation on connection status change.

If application developed is client, client should configure the endpoint which server can use while announcing to client. Client's port which will be listening to server's announcement, can be set using the API **EnableInboundConnection()** in UAClient object. Then client application can either accept or reject the incoming connection by implementing the class which is inherited by **IClientInboundConnectionListener_t** interface class.

After server initiated the connection and client accepted the connection, client will start with the Hello message to proceed with the usual service request and response messages as per the specification.

Server

User Authentication APIs

The server must implement the APIs provided by the module IUserPermissionsProvider_t to verify the user token provided by the client.

- **GetUserPermissions()** - This API is used to get the user permission for the given user token policy.

The application instance certificate has the public key (which is shared with other applications) and private key (used for encryption of the message). Private Key is used extensively by the security module to encrypt the messages. Private Key is password protected. To get the password and to unlock the private key so as to use it for encrypting the messages, the server developer must implement the API provided by ICertificatePrivateKeyPasswordProvider_t module.

- **GetPassword()** - This API is used to get the password.

When the client tries to connect to the server for the first time, the client's application certificate is rejected. TheServer developer has the control to decide, where the rejected certificate should be stored. To achieve this, the server developer should implement API provided by ICertificateRejectedListListener_t module.

- **CertificatedAddedToRejectedList()** - This API is used to store the rejected certificate in a specific location based on the certificate group.

Runtime Configuration APIs

- **ServerState** - This API is used to get the state of the server.

Server Capabilities

The server capabilities are limits that are exposed in the address space for clients to see which are enforced by the server to manage internal resources.

- **CapabilitiesMaxBrowseContinuationPoints()** - This API is used to get/set the maximum browse continuation points.

This API is for configuring the maximum number of continuation points supported by the browse service. The default value is 10.

Warning:

Continuation points require significant amounts of memory. For this reason, it is important not to enable this feature or make this value any larger than what the user's application requires.

- **CapabilitiesMaxQueryContinuationPoints()** - This API is used to get/set maximum query continuation points.
- **CapabilitiesMaxHistoryContinuationPoints()** - This API is used to get/set the maximum history continuation points.
- **CapabilitiesMaxArrayLength()** - This API is used to get/set the maximum array length.

This API is for configuring the maximum array length supported by the server. The default value is 256 bytes but the actual value is application specific.

- **CapabilitiesMaxStringLength()** - This API is used to get/set the maximum string length.

This API is for configuring the maximum UTF8 string length supported by the server. The default value is 4096 bytes but the actual value is application specific.

Operation Limits

The operation limits are limits that are exposed in the address space for clients to see which may be enforced by the server to manage internal resources.

• **OperationLimitsEnforce()**	-	This API is used to get/set the enforce operation limits. Set to TRUE to enforce limits. Enforcing limits protect against out of memory conditions but may affect interoperability with clients that do not observe these values. The default value is FALSE.

Warning:

In the event of enforcing limits, the user must test interoperability against all clients that are intended to interoperate with.

• **OperationLimitsMaxNodesPerRead()**	-	This API is used to get/set maximum nodes per read. Maximum nodes to read in a read service call. The default value is 100.
• **OperationLimitsMaxNodesPerHistoryReadData()**	-	This API is used to get/set maximum number of history read data. The default value is 100.
• **OperationLimitsMaxNodesPerHistoryReadEvents()**	-	This API is used to get/set maximum number of history read events. The default value is 100.
• **OperationLimitsMaxNodesPerHistoryUpdateData()**	-	This API is used to get/set maximum number nodes per history read update data. The default value is 100.
• **OperationLimitsMaxNodesPerHistoryUpdateEvents()**	-	This API is used to get/set maximum number of nodes per history update events.
• **OperationLimitsMaxNodesPerWrite()**	-	This API is used to get/set maximum number of nodes per write. The default value is 100.
• **OperationLimitsMaxNodesPerMethodCall()**	-	This API is used to get/set the maximum number of nodes per method call. The default value is 100.

- **OperationLimitsMaxNodesPerBrowse()** - This API is used to get/set maximum nodes per browse. The default value is 100.
- **OperationLimitsMaxNodesPerRegisterNodes()** - This API is used to get/set maximum nodes per register node. The default value is 100.
- **OperationLimitsMaxNodesPerTranslateBrowsePathsToNodeIds()** - This API is used to get/set maximum nodes per translate browse path node ids. The default value is 100.
- **OperationLimitsMaxNodesPerNodeManagement()** - This API is used to get/set maximum bodes per node management. The default value is 100.
- **OperationLimitsMaxMonitoredItemsPerCall()** - This API is used to get/set maximum monitored items per call. The default value is 100.

Services Configuration

The session service negotiates a revised session timeout with a client when the client connects to the server. These settings set a limit on the allowable range that the server accepts. If the client requests a timeout outside of these limits, then the server restricts the negotiated timeout to the appropriate limit.

- **MinRevisedSessionTimeoutInMs()** - This API is used to get/set minimum session timeout
- **MaxRevisedSessionTimeoutInMs()** - This API is used to get/set maximum session timeout

Warning:

- It is a good practice to make sessions relatively long-lived as this allows data collection to continue during a network transient. When connectivity is re-established, the client can login to the session and retrieve any data that was generated during the transient condition rather than having to create a new session and potentially losing data.
- However, sessions and secure channels are finite resources and it is important that they are released for other clients to use if they are no longer in use. Normally client’s close sessions on disconnect, but in the case of a client terminating a connection without closing the session, the session is required to timeout within a reasonable period.
- Additionally, the SDK recycles secure channels that have no active session associated with them. This gives a means of reusing both sessions and secure the channels that are inadvertently left open by a disorderly Client.

    - Keep session timeouts much smaller than secure channel timeouts to enable session and secure channel recycling if the user's application allows it.
    - The default values are 60 for both. But the required values are application specific.

- **MaxSessions() -** This API is used to get/set the maximum sessions

  This API is for configuring the maximum number of sessions that the server supports. This value must be equal to the number of concurrent client connections required.

Warning:

Sessions require significant amount of memory. For this reason, it is important not to make this value any larger than what the user's application requires.

- **MaxSubscriptionsPerSession()** - This API is used to get/set maximum subscription per session

  This API is for configuring the maximum number of subscriptions supported per session. Subscriptions are the server component that reports data changes over the wire to the client. Subscriptions execute at a cyclic rate and this cyclic rate does not affect the sample rate of variables by the server. (It affects only the rate at which they are reported).

  The server can support multiple monitored items sampling at different rates within a single subscription sampling at a slower rate where the server supports sample queues.

  If the server does not support sample queues and it is required to support multiple monitored items sampling at different rates, different subscriptions would be required to execute at the same rate as the contained monitored items are sampling.

  The number of subscriptions required is application specific.

Warning:

Subscriptions require significant amounts of memory. For this reason, it is important not to make this value any larger than what the user's application requires.

- **MinMaxKeepAliveCount()** - This API is used to get/set minimum for maximum keep alive count.
- **MaxMaxKeepAliveCount()** - This API is used to get/set maximum for maximum keep alive count.

The subscription service set negotiates a revised max keep alive count with a client when the client creates a subscription. These settings place bounds on the allowable range that the server accepts. If the client requests a count outside of these limits, then the server restricts the negotiated count to the appropriate limit.

The max keep alive count for a subscription defines how long the subscription will remain open in the absence of publishing requests from the client. The values for these limits are application specific. Unless the user has a specific reason to restrict the client in its choice of max keep alive count, it is recommended to keep these limits very wide.

- **MaxNotificationRetransmissionQueueSize()** - This API is used to get/set maximum retransmission queue size.

Clients create subscriptions to sample data where only the latest live value is required. In other cases, a client may wish to create a subscription where all sampled values require to be logged and do not want to lose any data. In this case, it may be helpful to enable a notification retransmission queue for subscriptions.

The notification retransmission queue retains all previously sent notification messages in a FIFO manner. In the event of a network transient, the client can reconnect to the server and query the queue to obtain any notifications it may have missed during the transient.

The notification retransmission queue size is application specific and may be set to 0.

Warning:

Notification retransmission queues require significant amounts of memory. For this reason, it is important not to enable this feature or make this value any larger than the application requires.

- **MaxPublishRequestsPerSession()** - This API is used to get/set the maximum publish request per session.

This API is for configuring the subscriptions work by a client issuing a publish request to the server. The server stores this request in a queue. The subscription executes at its cyclic rate and if data is available to report, the server disorganizes the request and sends a publish response containing data change notifications to the Client.

The server has the capacity to queue multiple publish requests internally to allow timely reporting over high latency networks. This allows it to serve periodic data to the Client at a faster rate than the latency of the network.

The maximum number of publish requests per session must be greater than the maximum number of subscriptions per session.

The number of publish requests per session required is application specific.

Warning:

Queuing publish requests requires significant amounts of memory. Hence, it is important not to make this value larger than what the user's application requires.

- **MaxBrowseRefsPerNode()** - This API is used to get/set Maximum browse references per node.

When the server responds to a browse request, it can limit the number of references included in the response and prompt the client to make another service call to get further references. This allows the server to limit the size of browse response messages and slows down the browse process as more messages are required.

A large server may set this value to 1000 and a very small server may set the value to 50. The actual value is application specific.

- **MaxMonitoredItems()** - This API is used to get/set the maximum monitored items.

This API is for configuring the maximum number of monitored items that the server can support across all sessions.

Warning:

Monitored Items require significant amounts of memory and CPU bandwidth. For this reason, it is important not to make this value any larger than what the user's application requires.

- **MaxMonItemQueueSize()** - This API is used to get/set the maximum sample queue length for monitored items.

This API is for configuring the maximum sample queue length for monitored items. Where a client is only interested in the latest live data value, the sample queue length can be set to 1. Where a client may be interested in logging all values and/or sampling monitored items at a slower rate than the containing subscription reports results, this value may be set to greater than 1. The actual value is application specific.

Warning:

Monitored item sample queues require significant amounts of memory. For this reason, it is important not to make this value any larger than the application requirements.

• **AddressSpace()**	- This API is used to get/set Address space.
• **IP addresses()**	- This API is used to get/set the IP address.
• **Hostnames()**	- This API is used to get/set the hostnames.
• **ApplicationURI()**	- This API is used to get/set the Application URI.
• **BuildInfo()**	- This API is used to get/set the build information.
• **LocaleIds()**	- This API is used to get/set the locale ids supported by the server**.**
• **TimeEmulationEnable()**	- This API is used to enable/disable time emulation.
• **UserPermissionsProvider()**	- This API is used to get/set user permission provider.

TCP Binary Stack Layer

- **TCPBinaryConnectionTimeoutInMs()**- This API is used to get/set connection timeout.

  This API is for configuring the allowed time between messages from the Client. The default value is 60 but the required value is application specific and depends on the expected message frequency for the application. The server closes the TCP Binary connection and corresponding TCP socket if this duration elapses with no messages from the client.

  Warning:

  TCP Binary connections and network sockets are finite resources. For this reason, it is important not to make this value too large as a misbehaving client may keep a connection open inadvertently and prevent other clients from using the resource.
- **DiscoveryEndpointEnable()** - This API is used to enable/disable the discovery endpoint.
- **DiscoveryEndpointTCPPort()** - This API is used to get/set discovery endpoint TCP port.
- **ServerEndpointTCPPort()**- This API is used to get/set server endpoint TCP port.

  The TCP port that the application is syncing to for incoming client connections. Port 4840 is the standard port.

- **TCPBinaryMaxMessageSize()-** This API is used to get/set maximum message size.

  This API is for configuring the maximum message size in bytes that the server accepts from a client. This must be at least as big as a single message chunk. Typical values are as follows:

  - 4MB for a server with lots of memory and large data sets.
  - 65kB for a mid-range server.
  - Just over 8kB for the smallest servers (the recommended size is 8200 bytes).

  For very small servers set this value equal to the message chunk size of 8200 bytes and set the maximum number of message chunks to 1.

- **TCPBinaryMaxMessageChunkSize()** - This API is used to get/set the maximum message chunk size.

  The maximum message chunk size is most significant when using security as the overall message chunked into pieces that are signed and optionally encrypted. Typical values are:

  - 65kB for a server with lots of memory and large data sets.
  - 8200 bytes for a mid-range to the small server.

- **TCPBinaryMaxChunkCount()** - This API is used to get/set maximum chunk count.

  This API is for configuring the maximum number of message chunks allowed while creating a message. This number multiplied by the message chunk size must be > = the maximum message size.

Warning:

⚠ Ensure by testing that the messages that the application requires to send and receive do not exceed the maximum message size. Additionally, the maximum message size should not be significantly larger than required as this increases the theoretical memory requirement of the server.

- **TCPBinaryMaxConnections()** - This API is used to get/set maximum connections.

  This API is for configuring the maximum number of TCP Binary connections that the server supports. This value must equal at least to the number of secure channels required plus 1.

Warning:

⚠ TCP Binary connections require significant amounts of memory. For this reason, it is important not to make this value any larger than what the user's application requires.

Secure Conversation Stack Layer

The secure channel layer negotiates a revised channel lifetime with a client when the client connects to the server. These settings place bounds on the allowable range that the server accepts. If the client requests a lifetime outside of these limits, then the server restricts the negotiated lifetime to the appropriate limit.

- **SecureChannelMinRevisedChannelLifetimeInMs()** - This API is used to get/set minimum revised channel life time.

- **SecureChannelMaxRevisedChannelLifetimeInMs()** - This API is used to get/set maximum revised channel life time.

Warning:

⚠

- When using security, the secure channel must renew the symmetric keys at the lifetime frequency. Renewing the keys requires the server to perform RSA private key operations and these are very slow CPU intensive operations. It is not necessary to renew the symmetric keys very often as all known brute force attacks on AES take a very long time to succeed. For example, a theoretical supercomputer that could check 50 *1018 AES keys per second would require about 3×1051 years to exhaust the 256-bit key space.
- The default values are from 1–24 hours and it is recommended not to alter these values.

- **SecureChannelMaxChannels()** - This API is used to get/set maximum secure channels.

This API is for configuring the maximum number of secure channels that the server supports. This value must equal at least the number of concurrent client connections required plus 1 extra channel for the integrated discovery endpoint.

Warning:

Secure channels require significant amounts of memory. For this reason, it is important not to make this value any larger than the application requires.

Security

• **SecurityCertificateDirectoryStoresPath()**	– This API is used to get/set the certificate directory store path.
• **GetApplicationInstanceCertificate()**	– This API is used to get the application instance certificate.
• **SendApplicationInstanceCertificateForSecurityPolicyNone()**	– This API is used to send the application instance certificate for security policy **None**.
• **AddEndpointUserTokenPolicy()**	– This API is used to add endpoint user token policy.
• **RemoveEndpointUserTokenPolicy()**	– This API is used to remove endpoint user token policy.
• **CloseEndpoint()**	– This API is used to close endpoint.
• **CloseEndpoints()**	– This API is used to close endpoints.
• **CloseAllEndpoints()**	– This API is used to close all endpoints.
• **IsEndpointOpen()**	– This API is used to check if the endpoint is open.
• **CertificateValidationOptions()**	– This API is used to get/set the certification validation option.
• **CertificatePrivateKeyPasswordProvider()**	– This API is used to get/set the certificate password provider.

• **CertificateRejectedListListener()**	– This API is used to get/set certificate rejected list listener.

Callbacks/Listeners

The following APIs help the server developer to configure the server and give callback or notification to the application layer on a certain operation.

• **RegisterNodesServiceTransactionListener()**	– This API is used to get/set transaction listener for node registration service.
• **UnregisterNodesServiceTransactionListener()**	– This API is used to get/set transaction listener for unregister node service.
• **RegisterNodeListener()**	– This API is used to get/set the Register node listener.
• **ReadServiceTransactionListener()**	– This API is used to get/set the transaction listener for read service.
• **WriteServiceTransactionListener()**	– This API is used to get/set the transaction listener for write service.
• **CallServiceTransactionListener()**	– This API is used to get/set the transaction listener for method call service.
• **MonitoredItemListener()**	– This API is used to get/set the listener for monitored item.
• **ChannelListener()**	– This API is used to get/set the listener for secure channel activities.
• **ServerStateListener()**	– This API is used to get/set listener for server state change.

Filters

• **PercentDeadbandDisable()**	– This API is used to enable/disable monitored item percent dead band filtering for all variables.
• **PercentDeadbandVectorsDisable()**	– This API is used to enable/disable monitored item percent dead band filtering for all vector variables.
• **PercentDeadbandFloatingPointDisable()**	– This API is used to enable/disable monitored item percent dead band filtering for all floating-point variables.
• **PercentDeadbandDoubleDisable()**	– This API is used to enable/disable monitored item percent dead band filtering for all double precision floating point variables.
• **PercentDeadbandFloatingPointVectors Disable()**	– This API is used to enable/disable monitored item percent dead band filtering for all floating-point vector variables.
• **PercentDeadbandDoubleVectorsDisable()**	– This API is used to enable/disable monitored item percent dead band filtering for all double precision floating point vector variables.
• **UseSinglePrecisionFloatForAllFilter Operations()**	– This API is used to enable/disable single precision float for all filter operation.

Profiles

• **ConfigureAsNanoDeviceServerProfile()**	– This API is used to configure the server as Nano device profile.

- **ConfigureAsMicroDeviceServerProfile()** – This API is used to configure the server as Micro device profile.
- **ConfigureAsEmbeddedServerProfile()** – This API is used to configure the server as embedded server profile.

Intervals

- **MinPublishingIntervalInMs()** - This API is used to get/set the minimum publishing intervals.
- **MinSamplingIntervalInMs()** - This API is used to get/set the minimum sampling intervals.
- **ServerCyclicRateInMs()** - This API is used to get/set the cycling rate.
- **ShutdownDelayInSeconds()** - This API is used to get/set server shutdown delay inseconds.
- **ShutdownDwellTimeInSeconds()** - This API is used to get/set server shutdown dwell time in seconds.

Events

- **MaxOperandsInListOperator()** – This API is used to get/set the maximum number of operands for the InList operator.
- **IsServerObjectOnlyEventNotifierNode()** – This API is used to get/set if the server object is only event notifier node.
- **IsServerApplicationReceivingEventNotificationEnabled()** – This API is used to get/set if the server application receives event notification enabled.
- **IsModelChangeEventsCompressionEnabled()** – This API is used to enable/disable the model change events compression.
- **IsSemanticChangeEventsCompressionEnabled()** – This API is used to enable/disable the semantic change events compression.
- **EventTypesActivationStatus()** – This API is used to get/set a 32-bit event activation status for each individual event type (refer to the header file "*event_type_activation_bits_defs.h*

” for more detailed bit activation definition of each event type).

EventHelper

EventHelper_t: This is a helper class to create events. It has the following APIs which can be used to create events and set specific properties.

• **CreateEventAndSetBaseEventProperties()**	– This API is used to create the event object based on the input and to set the BaseEventType properties.
• **SetAuditEventProperties()**	– This API is used to set the audit event properties for the audit event object as input.
• **CreateAuditSessionTypeEvents()**	– This API is used to create audit session type of events.
• **CreateAuditCertificateTypeEvents()**	– This API is used to create audit certificate type of events.
• **CreateProgressEvent()**	– This API is used to create the Progress Event instance.
• **GetDefaultSourceName()**	– This API is used to get default source name for an event.
• **GetDefaultMessageText()**	– This API is used to get the default message text for an event.
• **GetClientUserId()**	– This API is used to get the client UserId for Audit Session type events.
• **GetServerNodeId()**	– This API is used to get the NodeId of the Server object.

• **GetSecureChannelIdAsString()**	– This API is used to get the input Secure ChannelId as String.
• **GetCertificateThumbprint()**	– This API is used to get the thumbprint of the client certificate as input.
• **AddToLocalizableTextStored()**	– This API is used to add the input locale and text to the localizableTextStored. *LocalizableTextStored* is an out parameter which is used to pass the message parameter of any given event.

Address Space

Low Level Address Space API

Finding a node in address space

Nodes are looked up in the address space using the node ID which is a unique identifier for a node in the address space. Overloads of each Find() function are provided to allow the caller to filter results based on additional characteristics such as the security level of the current session/secure channel.

All Find() functions results in an intrusive reference counting boost-style smart pointer to the node. The caller must test the pointer before dereferencing it.

A node in the address space can be found with the help of following APIs:

• **FindNode()**	- This API is used to find a node in general.
• **FindViewNode()**	- This API is used to find a View node.
• **FindObjectNode()**	- This API is used to find an Object node.
• **FindObjectTypeNode()**	- This API is used to find an Object Type node.
• **FindVariableNode()**	- This API is used to find a Variable node.
• **FindVariableTypeNode()**	- This API is used to find a Variable Type node.
• **FindMethodNode()**	- This API is used to find a Method node.
• **FindDataTypeNode()**	- This API is used to find a Data Type node.
• **FindReferenceTypeNode()**	- This API is used to find a ReferenceType node.

Creating a node in the address space

A node can be created in the address space with the help of a node ID. Create methods do not look for duplicate node ID. If the user wants to avoid duplicate node ID, the caller must search for the duplicate node ID in the address space. All nodes created by the create methods will be hidden by default. The node must be marked as un-hide before address space searches and returns the node.

Any Create() APIs, returns an intrusive reference counting boost-style smart pointer to the node. The caller must test the pointer before dereferencing it.

Nodes can be created by the following APIs:

• **CreateView()**	- This API is used to create a View node.
• **CreateObject()**	- This API is used to create an Object node.
• **CreateObjectType()**	- This API is used to create an ObjectType node.
• **CreateVariable()**	- This API is used to create a Variable node.
• **CreateVariableType()**	- This API is used to create a VariableType node.
• **CreateMethod()**	- This API is used to create a Method node.
• **CreateDataType()**	- This API is used to create a DataType node.
• **CreateReferenceType()**	- This API is used to create a ReferenceType node.

Miscellaneous

The following APIs are used for specific operations as described below:

- **RemoveNode()**

  This function is used to remove the node from the address space.

  Warning:

  Removing the node does not remove the forward and/or inverse references. To completely remove the node from the memory before calling the RemoveNode() function, the reference(s) which has this node as target node has to be removed.

- **AddNodeAlias()**

  For servers that use string based node IDs, the RegisterNodes service may create alias Node IDs to speed up node lookup. This method adds the alias to the provided node.

  Warning:

  The value provided must fullfil several rules.

  - No non-constant references to the NodeId_t must exist, i.e., it must be immutable (to ensure thread safety).
  - The NodeId_t must be created dynamically using a SafeRefCount_t (to ensure thread safety).

    - The value pointer must be set.

- **RemoveNodeAlias()**

  This function is used to remove the alias from the node.

Node

Any object in the address space is considered as a node. Every node in the address space has attributes and references information. (All this information about a node is called Meta-data of a node). This information can be accessed and/or modified with the help of APIs provided with **INode_t** class.

Attribute APIs

The following APIs helps an application developer to access the attribute information of the node.

• **NodeId()**	-	This API is used to get the NodeId attribute.
• **NodeClass()**	-	This API is used to get the NodeClass attribute.
• **BrowseName()**	-	This API is used to get/set the BrowseName attribute.
• **DisplayName()**	-	This API is used to get/set the DisplayName attribute.
• **Description()**	-	This API is used to get/set the Description attribute.
• **WriteMask()**	-	This API is used to get/set the WriteMask attribute.
• **UserWriteMask()**	-	This API is used to get/set the UserWriteMask attribute.

Reference APIs

References of a node are stored with the help of the iterator. APIs are provided not only to create and delete a reference but also to access all the references associated with a node. Following are the APIs that can be used to access the reference information:

• **ReferencesGetConstIterator()**	-	This API is used to get the constant iterator for the node references.
• **ReferencesGetMutableIterator()**	-	This API is used to get a modifiable iterator for the node.
• **ReferencesRemoveAllReferences()**	-	This API is used to remove all the associated references for a node.
• **ReferencesRemoveReference()**	-	This API is used to remove a specific reference for a node.

- **ReferencesCreateReference()** - This API is used to create a reference for a node.

Special APIs

The following APIs help the application developer to take advantage of the customization feature in the SDK. The APIs are as follows:

- **Alias()**
    - For String based nodes, associating an alias help in faster lookup of the node. This API allows the application developer to get/set the alias information.
- **Hide()**
    - When a node is created in the address space, by default the node is hidden. Hidden nodes cannot be found using the provided Find() APIs. This API helps the application developer to hide/unhide a node.
- **IsVisible()**
    - This API is used to know whether the node is visible to a specific user.
- **Accept()**
    - A function used to safely downcast the node to its concrete type used by AddressSpaceUtilities_t::NodeDownCast().
- **RequiresSecureTransportLayer()**
    - Some Nodes should only be accessed over a secure connection. This API lets the application developer get/set the information whether the node is accessible over an insecure connection.
- **ApplicationContextHandle()**
    - This API allows the application developer to insert arbitrary, application-specific data into the node for easy retrieval on demand. This API should be used to get/set context handle.
- **SetAttributeAboutToBeReadListener()**
    - The application developer has the option to modify the attribute of the node just before the node attribute is about to read. By using this API, the application developer sets the listener. After setting a listener to a node, whenever a client tries to read the attribute of the node, the listener is called first so that application developer can change the value of the attribute just before it is read.
    - This API should be set with the thread-safe object of type INodeAttributeAboutToBeReadListener_t.

Warning:

- Once the listener is set, it is the responsibility of the application developer to remove the listener, if required.
- If the empty object is set, then the listener is removed from the node.

- **AttributeChangedListenerAdd()**
  - This API can be used to get the attribute changed information for a non-constant referenced node.
  - This API should be set with the thread-safe object of type INodeAttributeChangedListener_t.
- **AttributeChangedListenerRemove()**
  - This API is used to remove the attribute changed listener which is added with the help of AttributeChangedListenerAdd() API.
  - This API should be set with the thread-safe object of type INodeAttributeChangedListener_t.
- **ReferencesChangedListenerAdd()**
  - This API is used to get the reference changed information for a non-constant referenced node.
  - This API should be set with the thread-safe object of type INodeReferencesChangedListener_t.
- **ReferencesChangedListenerRemove()**
  - This API is used to remove the attribute changed listener which is added with the help of ReferencesChangedListenerAdd() API.
  - This API should be set with the thread-safe object of type INodeReferencesChangedListener_t.
- **EventListenerAdd()**
  - This API is used to add the event listener callback for a given node.
  - This API should be set with the thread-safe object of type INodeEventListener_t.
- **EventListenerRemove()**
  - This API is used to remove the event listener which is added with the help of EventListenerAdd() API.
  - This API should be set with the thread-safe object of type INodeEventListener_t.

IReference_t

An object of this class is used to get more detailed information for a node's reference. The following APIs allows the application developer to achieve the same.

• **TypeId()**	- This API is used to get the Reference Type ID.
• **Type()**	- This API is used to the ReferenceType node of the reference.
• **TargetNodeId()**	- This API is used to the target node ID of the reference.
• **TargetNode()**	- This API is used to get the target node of the reference.
• **IsInverse()**	- This API is used to get the reference inverse.
• **IsInView()**	- This API is used to get the reference in the specific view.
• **AddView()**	- This API adds reference to a view.
• **RemoveView()**	- This API removes the reference to a view.

INodeAttributeAboutToBeReadListener_t

An instance of this class is used for setting the attribute, about to read listener for a node. This class provides an API which is used to set the listener for an attribute and node. This class also provides an API to get the attribute mask information.

APIs are:

- **NodeAttributeAboutToBeRead()**
- **AttributeMask()**

INodeAttributeChangedListener_t

An instance of this class is used for adding/removing the attribute changed listener for a node. This class provides an API which is used to set/remove the listener for a certain attribute and node. This class also provides an API to get the attribute mask information.

APIs are:

- **NodeAttributeChanged()**
- **AttributeMask()**

INodeReferencesChangedListener_t

An instance of this class is used for adding/removing the reference changed listener for a node. This class provides API which is used to set/remove the listener for a certain type of node. APIs are:

- **NodeReferenceAdded()** - This API references the Added listener.
- **NodeReferenceRemoved()** - This API references a removed listener.

IUserPermissions_t

An object of this class is used to get/set the user permission for a node. The following APIs is used for the same.

- **GetVariableNodeUserAccessLevel()** - This API is used to get access level information of a variable node.
- **SetVariableNodeUserAccessLevel()** - This API is used to set access level information for a variable node.
- **GetNodeUserWriteMask()** - This API is used to get a user write mask information of a node.
- **SetNodeUserWriteMask()** - This API is used to set user write mask information for a node.
- **GetNodeUserVisibility()** - This API is used to get user visibility of a node.
- **SetNodeUserVisibility()** - This API is used to set user visibility for a node.
- **GetMethodNodeUserExecutable()** - This API is used to get a user executable information of a method node.
- **SetMethodNodeUserExecutable()** - This API is used to set a user executable information for a method node.

INodeEventListener_t

An instance of this class is used for adding/removing the event listener for a node. It provides an API which is used to set/remove the listener for the events and node.

- **EventNotify()** - This is an overloaded method which takes Object or View node as the first parameter.

IEventIdGenerator_t

An instance of this class is used for generating a unique event ID for an event.

- **GenerateEventId()** - This API is used to get a unique event ID in ByteString as out parameter.

IEventObjectFired_t

An instance of this class is used for dispatching an event to a target event notifier.

- **FireEventObject()** - This API is used to fire an event from the server stack.

UAServer_t

An instance of this class is used for integrating the UA Server implementation.

- **FireEvent()** - This API is used to fire an event from the server application instead of within the server stack.

Different Types of Nodes

The classes below describe the characteristics of different types of nodes that is created in the address space. Each of these nodes is derived from the INode_t. Hence all these nodes, support all the attributes and reference APIs as described in INode_t class.

IDataTypeNode_t

A class that represents a DataType node in the address space. The object of this class must be thread-safe.

This class provides the following API to get/set IsAbstract attribute information for a datatype node. If the DataType node is abstract, the instance of the DataTypeNode cannot be created in the address space.

- **IsAbstract()**
  IMethodNode_t

A class that represents a Method node in the address space. The object of this class must be thread-safe.

This class provides the following APIs to get/set attributes of the node which is specific to Method node.

- **Executable()** - This API is used to get/set the Executable attribute.
- **UserExecutable()** - This API is used to get/set the UserExecutable attribute.

This class provides some advanced APIs for application developer’s use of method node type. The APIs are:

- **SetMethodHandler()**
  - This API is used to execute a node by setting a method handler. Only one method can be registered for a node at any given time.
  - This API should be set with the thread-safe object of type INodeMethodHandler_t.

- **BeginMethodCall()**
    - This API is used to call the method of a node. If the method is executed successfully, CallMethodCompleted() is called by the method handlers.

INodeMethodHandler_t

The object of this class calls the method associated with the node with the help of CallMethodBegin(). If method call is a success, CallMethodCompleted() callback function is called.

ICallbackCallMethodCompleted_t

The object of this class is used for calling CallMethodCompleted() callback function. The application developer should create an object of this class if the application developer wants to make use of the method execution completion information. The API associated with this class is:

- **CallMethodCompleted()**

IObjectNode_t

A class that represents an Object node in the address space. The object of this class must be thread-safe. This class provides the following API to get/set EventNotifier attribute information for an Object node.

- **EventNotifier()**

IObjectTypeNode_t

A class that represents an ObjectType node in the address space. The object of this class must be thread-safe.

This class provides the following API to get/set IsAbstract attribute information for an ObjectType node. If the ObjectType node is abstract, the instance of the ObjectTypeNode cannot be created in the address space.

- **IsAbstract()**

IReferenceTypeNode_t

A class that represents a ReferenceType node in the address space. The object of this class must be thread-safe.

This class provides the following APIs to get/set the attributes of the ReferenceTypeNode. If the ReferenceTypeNode is abstract, the instance of the ReferenceTypeNode cannot be created in the address space. If the ReferenceTypeNode is symmetric, then creating the inverse reference is not necessary.

- **IsAbstract()** - This API is used to get/set the IsAbstract attribute.
- **Symmetric()** - This API is used to get/set the Symmetric attribute.
- **InverseName()** - This API is used to get/set the InverseName attribute.

IVariableNode_t

A class that represents the Variable node in the address space. The object of this class must be thread-safe.

This class provides the following APIs to get/set the attributes of the VariableNode.

- **Value()** - This API is used to get/set the Value attribute.
- **DataType()** - This API is used to get/set the DataType attribute.
- **ValueRank()** - This API is used to get/set the ValueRank attribute.
- **ArrayDimensions()** - This API is used to get/set the ArrayDimensions attribute.
- **AccessLevel()** - This API is used to get/set the AccessLevel attribute.
- **UserAccessLevel()** - This API is used to get/set the UserAccessLevel attribute.
- **MinimumSamplingInterval()** - This API is used to get/set the **MinimumSamplingInterval** attribute.
- **Historizing()** - This API is used to get/set the Historizing attribute.

This class provides APIs which the application developer implements to access the Variable Node effectively. The APIs are:

- **DisableSampling()**
  - For nodes with very large value attributed monitored item, sampling of the value attribute can be disabled to protect the platform resources from exhaustion. This API is used to get/set the monitored item value attribute sampling disabled information.
- **CanReadValueSynchronously()**
  - This API is used to determine if the node is read synchronously or not.
- **ValueAttributeReaderWriter()**

    - Variable node's value attribute is used to read and/or written synchronously/asynchronously. The application developer provides the latest value of the node with the help of value attribute reader and writer. If the value of the node to be read and/or written takes more time to get the value, then the application developer can set the reader writer for variable node and read and/or write the value of the node asynchronously in a different thread. The application developer can achieve the same with the help of this API.
    - This API should be set with the thread-safe object of type INodeValueAttributeReaderWriter_t.

- **BeginReadValue()**
    - This API is used to begin reading the value attribute of the variable node asynchronously.
    - On successful reading, reader calls ReadValueAttributeCompleted() callback function, so that application developer takes action after reading the value.

- **ReadValue()**
    - This API is used to read the value synchronously.
    - Before using this function, the user should check if the value attribute can be read synchronously with the help of CanReadValueSynchronously().

- **BeginWriteValue()**
    - This API is used to begin writing the value attribute of the variable node asynchronously.
    - On successful writing the value writer calls WriteValueAttributeCompleted() callback function so that application developer takes action after writing the value.

- **WriteValue()**

  This API is used to write the value synchronously.

INodeValueAttributeReaderWriter_t

The object of this class is used to set the user defined value attribute reader and writer for variable node with the help of the SetValueAttributeReaderWriter() function.

This class provides the following APIs to read the value synchronously/asynchronously. This class also provides the information if the read value attribute object is accessed synchronously or not.

• **ReadValueAttributeBegin()**	- This API is used to read the value attribute asynchronously.
• **ReadValueAttribute()**	- This API is used to read the value attribute synchronously.

• **CanReadValueSynchronously()**	- This API is used to know if the read value attribute object can be accessed synchronously or not.
• **WriteValueAttributeBegin()**	- This API is used to write the value attribute asynchronously.
• **WriteValueAttribute()**	- This API is used to write the value attribute synchronously.
• **CanWriteValueSynchronously()**	- This API is used to know if the write value attribute object is accessed synchronously or not.

ICallbackNodeValueAttributeAccessCompleted_t

The object of this class is used to call ReadValueAttributeCompleted() after successful completion of the read operation and WriteValueAttributeCompleted() after successful completion of the write operation.

IVariableTypeNode_t

A class that represents the VariableType node in the address space. The object of this class must be thread-safe.

This class provides the following APIs to get/set the attributes of the VariableTypeNode. If the VariableTypeNode is abstract, then the instance of the node should not be created.

• **Value()**	- This API is used to get/set the Value attribute.
• **DataType()**	- This API is used to get/set the DataType attribute.
• **ValueRank()**	- This API is used to get/set the ValueRank attribute.
• **ArrayDimensions()**	- This API is used to get/set the ArrayDimensions attribute.
• **IsAbstract()**	- This API is used to get/set the IsAbstract attribute.

ViewNode_t

A class that represents the View node in the address space. The object of this class must be thread-safe.

This class provides the following APIs to get/set the attributes of the ViewNode.

• **ContainsNoLoops()**	- This API is used to get/set the ContainsNoLoops attribute.
• **EventNotifier()**	- This API is used to get/set the EventNotifier attribute.

Other Modules in Server

IIterator_t

This module is needed for iterator list support for the SDK. The APIs that require to be implemented are as follows:

- **First()** - This API is used to get the first item in the iterator list.
- **Next()** - This API is used to get the next item in the iterator list.
- **Current()** - This API is used to get the current item in the iterator list.
- **RemoveCurrent()** - This API is used to remove the current item from the iterator list.

IAllocator_t

This module is needed for providing the memory information used by SDK and allocating the memory in the provided buffer. The APIs required to be implemented are as follows:

- **Malloc()** - This API is used to allocate the memory.
- **Free()** - This API is used to free the memory.
- **MemoryInfo()** - This API is used to get the memory usage information.

IThreadPool_t

This module is required for SDK to run in the multi-threaded platform. If the SDK is configured for the Multi-threaded platform, then APIs provided by this module require to be implemented. The APIs that is required to be implemented are as follows:

- **QueueToThreadPool()** - This API is used to queue the runnable object.
- **Operate()** - This API is used to operate a runnable object.

IRunnable_t

This module is required by the SDK to create a runnable object. The API that requires to be implemented is given below:

- **Run()** - This API is used to execute the object.

IRefCount_t

SDK implements the smart pointers internally to make sure that memory management is achieved with the help of object’s lifetime. This module is used to achieve the reference counting of the object when it is created.

The APIs that require to be implemented for this module are as follows:

- **AddReference()** - This API is used to add the reference.
- **RemoveReference()** - This API is used to remove the reference.
- **IsThreadSafe()** - This API is used to check if the object is thread-safe.

Access Types

The Specifications specifies three types of Access types.

- Data Access Types (DA)
- Events, Alarms and Conditions (AC)
- History Access Types (HA)

Data Access APIs

Refer to the **Data Access** section for data access related information.

All data access types are implemented with the assisstance of the helpers. Each Data access types is created with the following APIs:

CreateWithNumericId()	- This API is used to create a Numeric ID.
CreateWithStringId()	- This API is used to create a String ID.
CreateWithGuidId()	- This API is used to create a Guid ID.
CreateWithOpaqueId()	- This API is used to create a Opaque ID.

Note:

BaseDataVaraible type, Object type, Methods and the above methods is used to create the nodes.

Events, Alarms and Conditions APIs

Refer to **Events, Alarms and Conditions** section for more information.

IAttributeAccess_t

An instance of this class acts as an event instance. This class is used to fetch the attribute values of an event.

Different IAttributeAccess APIs are mentioned below:

IsOfType()

This API is used to get the typeId. The API is as follows:

IsOfType(const uint32_t typeId, uint16_t namespaceIndex)const

This API returns true if the event instance is type given by the input type ID, else it returns false. The *typeId* parameter indicates the input event instance. The *namespaceIndex* parameter indicates the reference input to the name space index.

AttributeValue()

These APIs are used to get the attribute values of the event attribute. The APIs are as follows:

- **AttributeValue(const uint32_t & typeId,const ArrayUA_t < QualifiedName_t > & browsePath,const uint32_t & attributeId,const String_t & indexRange,Status_t & result)const**

  This API returns the constant reference value of the requested attribute as result. The *typeId* parameter indicates the input type of the instance. The *browsePath* parameter indicates the input path from the instance to the node which defines the attribute. The *attributeId* is the input parameter to return. The parameter *indexRange* is the input parameter which indicates the sub-set of an array value to return. *Result* parameter indicates the output that publishes the operation status as result.

- **AttributeValue(const uint32_t & typeId,const ArrayUA_t < QualifiedName_t > & browsePath,const uint32_t & attributeId,const String_t & indexRange,Status_t & result)**

  This API returns the non-constant reference value of the requested attribute as result. The *typeId* parameter indicates the input type of the instance. The *browsePath* parameter indicates the input path from the instance to the node which defines the attribute. The *attributeId* is the input parameter to return. The parameter *indexRange* is the input parameter which indicates the sub-set of an array value to return. *Result* parameter indicates the output that publishes the operation status as result.

- **AttributeValue(const uint32_t & typeId,const ArrayUA_t < QualifiedName_t > & browsePath,const uint32_t & attributeId,const String_t & indexRange,const ArrayUA_t < String_t > & locales,**

Status_t & result)const)

This API returns the constant reference value of the requested attribute as result. The *typeId* parameter indicates the input type of the instance. The *browsePath* parameter indicates the input path from the instance to the node which defines the attribute. The *attributeId* is the input parameter to return. The parameter *indexRange* is the input parameter which indicates the sub-set of an array value to return. The *locales* parameter is the list of input locales from the current session. *Result* parameter indicates the output that publishes the operation status as result.

- **AttributeValue(const uint32_t & typeId,const ArrayUA_t < QualifiedName_t > & browsePath,const uint32_t & attributeId,const String_t & indexRange,const ArrayUA_t < String_t > & locales Status_t & result)**

This API returns the non-constant reference value of the requested attribute as result. The *typeId* parameter indicates the input type of the instance. The *browsePath* parameter indicates the input path from the instance to the node which defines the attribute. The *attributeId* is the input parameter to return. The parameter *indexRange* is the input parameter which indicates the sub-set of an array value to return. The *locales* parameter is the list of input locales from the current session. *Result* parameter indicates the output that publishes the operation status as result.

SubscriptionId()

This API is used to get the subscription ID for the given condition which is triggered from the conditionRefresh method. The APIs are as follows:

- **SubscriptionId(void)**
- **SubscriptionId(void)const**

These APIs return valid subscription ID from the conditionRefresh method if the data object is a condition, else it returns null.

Note:

This API is only for internal use.

MonitoredItemId()

This API is used to get the subscription ID and the monitored item ID for the given condition which is triggered from the conditionRefresh2 method. The APIs are as follows:

- **MonitoredItemId(void)**
- **MonitoredItemId(void)const**

These APIs return valid subscription ID and monitored item ID from the conditionRefresh2 method if the data object is a condition, else it returns null.

Note:

This API is only for internal use.

INodeEventListener_t

An instance of this class is used to register events for a given node. The implementation effort of this interface must be thread safe.

The INodeEventListener API is mentioned below:

EventNotify()

This API is used to callback event listeners for object nodes and view nodes. The APIs are as follows:

- **EventNotify(IObjectNode_t & node,IntrusivePtr_t < IAttributeAccess_t > & eventAttributes)**
- **EventNotify(IViewNode_t & node,IntrusivePtr_t < IAttributeAccess_t > & eventAttributes)**

The *node* parameter indicates the input that describes the intrusive pointer of IObjectNode_t or IViewNode_t that is used to get the value. The input *eventAttribute* parameter indicates the input that describes the intrusive pointer of IAttributeAccess_t instance of the event.

IEventIdGenerator_t

An instance of this class is used to generate a unique event ID.

The IEventIdGenerator API is mentioned below:

GenerateEventId()

This API is used to generate the event ID. The API is as follows:

GenerateEventId(IntrusivePtr_t < ByteString_t > & eventId)

This API returns the operation status as result. The *eventId* parameter indicates the output that describes the generated event ID.

IEventDispatcher_t

An instance of this class is used to fire an event.

The IEventDispatcher API is mentioned below:

FireEventObject()

This API is used to fire an event. The API is as follows:

FireEventObject(

IntrusivePtr_t < IAttributeAccess_t > eventObject,

IntrusivePtr_t < NodeId_t > eventSourceNode)

This API returns the operation status as result. The *eventObject* parameter indicates the input that describes the event object to be fired. The input *eventSourceNode* parameter indicates the node ID of the event source node.

IContentFilter_t

An instance of this class is used to configure the filtering criteria for the events.

The IContentFilter API is mentioned below:

EvaluateFilterElement()

This API is used to process the filter element and obtain the result of evaluating the filter against the provided node. The API is as follows:

EvaluateFilterElement(uint32_t element,const INode_t & node,IntrusivePtr_t < IAttributeAccess_t > & eventAttributes,IntrusivePtr_t < IServerConfiguration_t > configuration,IntrusivePtr_t < BaseDataType_t > & evaluatedResult,IntrusivePtr_t < ContentFilterElementResult_t > & elementResult, bool diagnosticsRequired)

This API returns a status good if the filter is applied correctly, else it returns a bad error code. The *element* parameter indicates the index to be processed. The *node* parameter indicates the input to pass through the filter. The *eventAttributes* parameter indicates the event instance. The input parameter *configuration* is the instance of server configuration. The *evaluatedResult* parameter describes the result of the evaluation. *ElementResult* is the result of the given content filter element. *DiagnosticsRequested* is the input flag to get the diagnostic information in the result structure.

IAuditEventListener_t

An instance of this class provides interfaces to listen for audit events generated in the server stack.

The IAuditEventListener API is mentioned below:

AuditEventOccured()

This API is called when an audit event is fired. The API is as follows:

AuditEventOccured(IntrusivePtr_t < IAttributeAccess_t > & auditEvent)const

IAuditEventAttributeLogger_t

An instance of this class is used to log attributes of the audit events.

The IAuditEventAttributeLogger API is mentioned below:

LogAttribute()

These APIs are used to audit event logger. The API is as follows:

- **LogAttribute(IntrusivePtr_t < BaseDataType_t > attribute)const**
- **LogAttribute(const char * data)const**

The *attribute* parameter indicates the input which describes the attribute value to be printed on the console or in the LogFile. The *data* parameter indicates the input which describes the data value to be printed on the console or in the LogFile.

ICallbackEventLogged_t

An instance of this class is used to log the events.

The ICallbackEventLogged API is mentioned below:

LogEvent()

This API is used to log the event. The API is as follows:

LogEvent(

IntrusivePtr_t < IAttributeAccess_t > eventObject,

String_t & eventRecordId)

This API returns the operation status as result. The *eventObject* parameter indicates the input that describes the event object to be logged. The output *eventRecordId* parameter indicates the string which represents the event record ID.

Default Events, Alarms and Conditions Properties (default_event_properties_t.h)

This file has the definitions for default macros used for events, alarms and conditions. The default properties can be modified in this file based on the application layer requirements.

IConditionRegistrar_t

An instance of this class is used to maintain the list of conditions created.

The IConditionRegistrar APIs are mentioned below:

Register()

This API is used to register the condition with the condition registrar. The API is as follows:

Register(IntrusivePtr_t < ICondition_t > &condition)

This API returns the operation status as result. The *condition* parameter indicates the intrusive pointer of the type ICondition_t that requires to be registered.

UnRegister()

This API is used to unregister the condition with the condition registrar. The API is as follows:

UnRegister(IntrusivePtr_t < ICondition_t > &condition)

This API returns the operation status as result. The *condition* parameter indicates the intrusive pointer of the type ICondition_t that requires to be unregistered.

UnRegisterAll()

This API is used to unregister all the conditions in the condition registrar. The API is as follows:

UnRegisterAll(void)

This API returns the operation status as result.

GetCondition()

This API is used to get the condition from the condition registrar based on the input condition ID. The API is as follows:

- **GetCondition(const NodeId_t & conditionId, Status_t & result)const**

- **GetCondition(const NodeId_t & conditionId, Status_t & result)**

This API returns the condition instance as result, else it returns empty instance. The *conditionId* parameter indicates the input condition ID for the condition. The output parameter *result* describes the operation status.

CheckConditionIdIsAvailable()

This API is used to check if the input condition ID exists in the address space and condition registrar. The API is as follows:

CheckConditionIdIsAvailable(const NodeId_t & conditionId)
The *conditionId* parameter indicates the input condition ID for the condition. It returns good if the input condition ID is unique and does not exists in the address space and condition registrar, else it returns *BadNodeIdExists*.

ICondition_t

This is an interface class which should be implemented by all the conditions.

Different ICondition APIs are mentioned below:

ConditionId()

This API is used to get the condition ID. The API is as follows:

- **ConditionId(void)**

This API returns a unique condition ID as result.

MountToAddressSpace()

This API is used to mount the condition to the address space. The API is as follows:

- **MountToAddressSpace(const String_t & browseName,uint32_thasTypeDefinitionId,NodeId_t & parentNodeId)**

This API returns the operation status as result. The *browseName* parameter indicates the input and is used get the browseName of the condition. The parameter *hasTypeDefinitionId* is the input parameter which indicates the typeDefinitionId of the condition. The parameter *parentNodeId* describes the nodeId of the parent node in the address space under which the condition is going to be mounted.

UnMountFromAddressSpace()

This API is used to unmount the condition from the address space. The API is as follows:

- **UnmountFromAddressSpace(void)**

This API returns the operation status as result.

RelocateInAddressSpace()

This API is used to mount the condition to the address space. The API is as follows:

- **RelocateInAddressSpace(const String_t & browseName, uint32_t hasTypeDefinitionId, NodeId_t & newParentNodeId)**

This API returns the operation status as result. The *browseName* parameter indicates the input and is used get the browseName of the condition. The parameter *hasTypeDefinitionId* is the input parameter which indicates the typeDefinitionId of the condition. The parameter *newparentNodeId* describes the node ID of the new node under which the condition needs to be relocated.

ConditionData()

This API is used to get the condition data instance. The API is as follows:

- **ConditionData(void)**

This API returns the instance of condition data as result.

GetEventData()
This API is used to get the copy of the event data to be fired to the client. The API is as follows:

- **GetEventData(IntrusivePtr_t < IAttributeAccess_t > & data, IntrusivePtr_t < NodeId_t > & sourceNodeId)**

This API returns the operation status as result. The *data* parameter indicates the output and is used to get the browseName of the condition. The parameter *sourceNodeId* is the output parameter which indicates the source node ID associated with the event.

IsMountedToAddressSpace()

This API is used to determine if the condition is mounted in the address space or not. The API is as follows:

- **IsMountedToAddressSpace(void)**

This API returns true if the condition is mounted in the address space, else it returns false.

IConditionStateUpdateHandler_t

This is a callback interface included in the ICondition_t to allow condition variable and state variable changes to be notified, and to allow the condition to determine if an event notification is required.

Different IConditionStateUpdateHandler APIs are mentioned below:

BeginConditionStateUpdate()

This API is used to group multiple condition state in a single notification before calling *ConditionStateUpdated()*. The API is as follows:

- **BeginConditionStateUpdate(void)**

Every call to this API must be matched with a call to *EndConditionStateUpdate()*. BeginConditionStateUpdate/EndConditionStateUpdate blocks is built recursively, and all updates made inside a block is sent as a single notification at the end.

FireEventNotification()

This API is called to take a copy of the current event information and post it to any subscribed UA clients through monitored items. The API is as follows:

- **FireEventNotification(void)**

This API returns the operation status as result.

ConditionStateUpdated()

This API is called if a condition variable member of the class has been changed. If it is called inside of a *BeginConditionStateUpdate*/*EndConditionStateUpdate* pair, the event notification is sent when all updates are completed and the final EndConditionStateUpdate call is made. If this method is called outside of a *BeginConditionStateUpdate*/*EndConditionStateUpdate* block, then an event notification is sent immediately. The API is as follows:

- **ConditionStateUpdated(void)**

This API returns the operation status as result.

EndConditionStateUpdate()

This API is used to end the condition update. The API is as follows:

- **EndConditionStateUpdate(void)**

This API returns the operation status as result.

BaseCondition_t

The *BaseCondition* defines all general characteristics of a condition.

This class implements ICondition_t and provides the default implementations for all the required methods.

All action commands that *update state* or *condition variables* of the event must call the implementation of *IConditionStateUpdatedHandler_t::CondtionStateUpdated* as a final action.

This class is sub classed by the user developer to produce final condition types.

Different BaseCondition APIs are mentioned below:

EvaluateState()

This API is used to evaluate the current state of the condition. The API is as follows:

- **EvaluateState(uint32_t methodId)**

This API returns the operation status as result. The input parameter *methodId* indicates the type ID of the method from which the *EvaluateState* is called.

InitialiseConditionData()

This API is used to initialize the data object. The API is as follows:

- **InitialiseConditionData(IntrusivePtr_t < ConditionTypeData_t > conditionData_,**

  uint32_t conditionClassId, String_t & conditionName,

  bool isAllocated = false)

This API returns the operation status as result. The input parameter *conditionData* indicates the condition data object. The parameter *conditionClassId* is used to assign a condition to a conditionClass. The parameter *conditionName* is used to identify the condition instance of the event originated. Parameter *isAllocated* is used to allocate conditionData if the status is false, else it assigns the input data object to the member variable.

GetStandardNodeId()

This API is used to return the type ID of the input node if condition is mounted in the address space. The API is as follows:

- **GetStandardNodeId(const INode_t & node)**

This API returns the type ID of the node if the condition is mounted to the address space, else it returns zero as result. The input parameter *node* indicates the input node object.

GetMethodNodeIdFromStandardNodeId()

This API is used to get the method node based on the standard type ID of the method. The API is as follows:

- **GetMethodNodeIdFromStandardNodeId(uint32_t standardNodeId,IntrusivePtr_t <NodeId_t > & methodId)**

This API returns the operation status as result. The input parameter *standardNodeId* indicates the type ID of the node. The output parameter *methodId* describes the node ID of the method.

OnDisable()

This API is called from CallMethodBegin and is intended to be used as hooks by the user developers for getting a notification of the change of the condition. The API is as follows:

- **OnDisable(const IUserPermission_t & userPermissions)**

This API returns the operation status as result. The input parameter *userPermissions* indicates an object used to identify a specific user and to authenticate that user's access permissions.

OnEnable()

This API is called from CallMethodBegin and is intended to be used as hooks by the user developers for getting a notification of the change of the condition. The API is as follows:

- **OnEnable(const IUserPermission_t & userPermissions)**

This API returns the operation status as result. The input parameter *userPermissions* indicates an object used to identify a specific user and to authenticate that user's access permissions.

OnAddComment()

This API is called from CallMethodBegin and is intended to be used as hooks by the user developers for getting a notification of the change of the condition. The API is as follows:

- **OnAddComment(const IUserPermissions_t & userPermissions, IntrusivePtr_t < constCallMethodRequest_t > & requestParameters)**

This API returns the operation status as result. The input parameter *userPermissions* indicates an object used to identify a specific user and to authenticate that user's access permissions. *RequestParameters* parameter describes the requested input parameter.

BeginConditionStateUpdate()

This API is used to group multiple condition state updates together in a single notification. The API is as follows:

- **BeginConditionStateUpdate(void)**

FireEventNotification()

This API is used to take a copy of the current event information and post it to any subscribed UA clients through monitored items. The API is as follows:

- **FireEventNotification(void)**

This API returns the operation status as result.

ConditionStateUpdated()

This API is called when there is a change in the condition state. If it is called inside of a *BeginConditionStateUpdate/EndConditionStateUpdate* pair, the event notification is sent when all updates are completed and the final *EndCOnditionStateUpdate* call is made. If this method is called outside of a *BeginConditionStateUpdate/EndConditionStateUpdate* block, then an event notification is sent immediately. The API is as follows:

- **ConditionStateUpdated(void)**

This API returns the operation status as result.

EndConditionStateUpdate()

This API is called when there is an end to the condition state update. The API is as follows:

- **EndConditionStateUpdate(void)**

This API returns the operation status as result.

EvaluateQualityFromEnabledState()

This API is used to return the quality based on the EnabledState. The API is as follows:

- **EvaluateQualityFromEnabledState(IntrusivePtr_t < ConditionVariableType_t > & quality_)**

This API returns the operation status as result. The output parameter *quality* describes the quality to be updated.

UpdateCommentState()

This API is used to update the comment state. The API is as follows:

- **UpdateCommentState(ByteString_t & eventId,LocalizedText_t & comment)**

This API returns the operation status as result. The input parameter *eventId* indicates the ID of the event to which comment is to be added. *Comment* parameter indicates the input comment to be added.

MayFireEventNotification()

This API is used by the *FireEventNotification* API to check if the notifications are required to send. The default implementation provided by BaseCondition_t checks whether the condition is enabled or not. This is overridden by the subclasses to provide more complex rules.

For example, The AlarmCondition_t class overrides this to support use of the suppressed state to block event notifications. The API is as follows:

- **MayFireEventNotification(void)**

This API returns true if the event notifications are allowed, else it returns false.

Initialise()

This API is used to create condition and update in the condition register if the condition is created successfully. The API is as follows:

- **Initialise(IntrusivePtr_t < IServerConfiguration_t > configuration_, NodeId_t &**

  conditionId_,uint32_t conditionClassId,String_t & conditionName)

This API returns the operation status as result. The input parameter *configuration* indicates the server configuration. *ConditionId* parameter indicates the ID of the condition. The parameter *conditionClassId* describes the input class ID of the condition. The parameter *conditionName* identifies the condition instance from where the event originated.

InitialiseEnabledStateLocalizationValues()

This API is used to set the localization values for enabled state.

Note:

Ensure that the condition is initialized before calling this API. The API is as follows:

- **InitialiseEnabledStateLocalizationValues(IntrusivePtr_t < ILocalizableText_t >**

  enabledStateTrueState, IntrusivePtr_t <

  ILocalizableText_t > enabledStateFalseState)

This API returns the operation status as result. The input parameter *enabledStateTrueState* describes the list of true values for the enabledState. The parameter *enabledStateFalseState* describes the list of false values for the enabledState.

RegisterCondition()

This API is used to register the condition with the condition registrar. The API is as follows:

- **RegisterCondition(NodeId_t & conditionId_)**

This API returns the intrusive pointer to the condition ID as result. The input parameter *conditionId* indicates the ID of the condition.

ConditionId()

This API is used to get the condition ID. The API is as follows:

- **ConditionId(void)**

This API returns the intrusive pointer to the condition ID as result.

ConditionNode()

This API is used to get the condition node from the address space. The API is as follows:

- **ConditionNode(Status_t & result)const**

This API returns the intrusive pointer to the object node as result. The output parameter *result* describes the operation status.

IsEnabled()

This API is used to check whether the condition is enabled. The API is as follows:

- **IsEnabled(void)**

This API returns true if the condition is enabled, else it returns false.

Disable()

This API is used to change a condition instance to the disabled state. The API is as follows:

- **Disable(void)**

This API returns the operation status as result.

Enable()

This API is used to change a condition instance to the enabled state. The API is as follows:

- **Enable(void)**

This API returns the operation status as result.

AddComment()

This API is used to apply a comment to a specific state of a condition instance. The API is as follows:

- **AddComment(ByteString_t & eventId,LocalizedText_t & comment)**

This API returns the operation status as result. The input parameter *eventId* indicates the ID of the event to which comment should be added. The parameter *comment* indicates the input comment to be added.

MountToAddressSpace()

This API is used to mount the condition in the address space. The API is as follows:

- **MountToAddressSpace(const String_t & browseName, uint32_t hasTypeDefinitionId, NodeId_t & parentNodeId)**

This API returns the operation status as result. The input parameter *browseName* indicates the browse name of the condition. The parameter *hasTypeDefinitionId* indicates the type definition reference of the condition. The input parameter *parentNodeId* describes the new parent ID.

SetMethodListener()

This API is used to set the method handler for the callback listener. The API is as follows:

- **SetMethodListener(IntrusivePtr_t < IBaseConditionMethodListener_t > value)**

This API returns the operation status as result. The input parameter *value* indicates the callback listener instance.

UnmountFromAddressSpace()

This API is used to unmount the condition from the address space. The API is as follows:

- **UnmountFromAddressSpace(void)**

This API returns the operation status as result.

RelocateInAddressSpace()

This API is used to relocate the condition in the address space. The API is as follows:

- **RelocateInAddressSpace(const String_t & browseName, uint32_t hasTypeDefinitionId, NodeId_t & newparentNodeId)**

This API returns the operation status as result. The input parameter *browseName* indicates the browse name of the condition. The parameter *hasTypeDefinitionId* indicates the type definition reference of the condition. The input parameter *newparentNodeId* describes the new parent ID.

IsMountedToAddressSpace()

This API is used to find if the condition is mounted in the address space. The API is as follows:

- **IsMountedToAddressSpace(void)**

This API returns true if the condition is mounted in the address space, else it returns false.

ConditionData()

This API is used to get the condition data instance. The API is as follows:

- **ConditionData(void)**

This API returns the condition data instance as result.

Service Call Listener Interface

The SDK provides a listener interface which allows the server application to see a subset of the incoming service call requests. The purpose of the call is to allow the application to defer processing of that request until it is ready. An example use case would be where the server address space is modelled using a lazy evaluation approach and at any given time may not contain an up to date model of all nodes that are logically present in the address space.

To use the feature, implement the listener interface, **IServiceCallListener**_t and register it with the server configuration. The listener will now be called for each service call request that may result in the client interacting for the first time with nodes in the address space. Taking the browse service as an example:

```
  virtual void BrowseRequestReceived(

    uint32_t requestId,

    IntrusivePtr_t<const BrowseRequest_t> request,

    IntrusivePtr_t<ICallbackServiceCall_t> completedCallback,

    Command_t& command) = 0;
```

The **requestId** argument is a unique identifier provided by the SDK to represent this service call in the service call listener interface. This can be used by the application to track subsequent completion of the service call handling within the SDK. When the service call processing is subsequently completed by the SDK the **IServiceCallListener_t::ServiceCallCompleted** method will be called with the relevant **requestId**. If the application does not wish to track service call completion, then this argument can be ignored. For HistoryRead and browse service calls, ServiceCallCompleted() will only be called after all continuation points associated with the request have been released.

The **request** argument is the actual BrowseRequest received from the UA client.

The **completedCallback** is an SDK callback object that the application can retain if it wants to defer the service call until it has updated its address space model.

The **command** output argument tells the SDK whether the application wants to defer the call processing. If the application returns **IServiceCallListener_t::Command_t::COMMAND_DEFER** then the SDK will not process the service call until the application has called **ICallbackServiceCall_t::Continue**. If the application returns **COMMAND_PROCESS** then the service call will be immediately processed by the SDK and the application can disregard the **completedCallback** argument.

Application Responsibilities

Where the application elects to defer a service call it must subsequently call **ICallbackServiceCall_t::Continue** to allow the call to be completed. Where the client has specified a TimeoutHint in the header of the service call the application is responsible for choosing to honour the timeout. The SDK will not process the service call and return a service response until the application has instructed it to do so.

CreateMonitoredItem Service Call

Monitored items can only be created via the CreateMonitoredItems service call but there are many ways a monitored item can be destroyed, including, a DeleteMonitoredItems service call, A DeleteSubscriptions service call, a CloseSession service call, a session or subscription timing out, etc. For this reason the **IServicecallListener_t::CreateMonitoredItemsRequestReceived** method should only be used in conjunction with the **IMonitoredItemListener_t** as this combination allows the application to track when monitored items are created and destroyed.

Client

The Client part of SDK is a set of cross-platform data types, structures and interfaces to API methods and UA concept related objects which user developers can utilize to add OPC UA functionalities to their OPC UA client applications.

IClientNotificationCallback_t

The UA Client application should realize this callback interface if subscriptions and monitoring are used. The Client part of SDK provides a helper class to manage the publishing and notifications which is BasicClientNotificationManager_t. To get the notifications of the monitored items in subscriptions, the instance of IClientNotificationCallback_t needs to register into BasicClientNotificationManager_t and notification messages will be dispatched to the client application by the callback method OnNotification().

OnNotification()

This is the callback API method which allowes user developer to get the notification messages and process them.

BasicClientNotificationManager_t

An instance of this class issues one or more publish requests per subscription on the initiated session and acknowledges the Publish response. The notification messages from the publish or republish responses will be automatically dispatched by IClientNotificationCallback_t:: OnNotification(). This class provides the simplest possible client publishing and republishing management system. It is multi-thread safe and the implementation effort required is very minimal.

Different BasicClientNotificationManager APIs are mentioned below:

- **Initialise()**
- **Session()**
- **StartPublishing()**
- **SendPublishRequest()**
- **SendRepublishRequest()**
- **Stop()**
- **Shutdown()**

Initialise()

This API is used to initialize the instance. The API is as follows:

- **Initialise(IntrusivePtr_t < IClientSession_t > session, IntrusivePtr_t < IClientCallback_t > onComplete)**

This API returns the operation status as result. The *session* is used for managing publishing and for callbacks. The parameter *onComplete* describes the callback reference function when the operation is complete.

Session()

This API is used to get the session instance.

StartPublishing()

Once UA client has subscription and monitored items created, this method can be used to send publish requests for each subscription. StartPublishing() is called in the Initialise().

SendPublishRequest()

In some use cases, after the basic client notification manager is initialized (publishing is started), new subscriptions with monitored items still can be created. In this case, SendPublishRequest() can be called for the new created subscription.

SendRepublishRequest()

Republishing is managed in the basic client notification manager as well as publishing. That means, user developer can use SendRepublishRequest() to send republish request if needed and the response is handled.

Stop()

This API is used to stop sending requests to the server and wait for the pending requests to complete. The API is as follows:

- **Stop(IntrusivePtr_t < IApplication_t > const & app)**

This API returns the operation status as result. The *application* parameter indicates the input and is used by the application instance for single-thread case to drive the poll loop.

Shutdown()

Before exiting from the UA client application, Shutdown() must be called because it releases resources which are occupied by basic notification manager.

PublishResponseHandler_t

Different PublishResponseHandler APIs are mentioned below:

- **Initialise()**
- **OnCompleted()**

Initialise()

This API is used to initialize the basic notification manager. The API is as follows:

- **Initialise(BasicClientPublishManager_t * a_parent)**

This API returns the operation status as result. The *parent* parameter specifies the reference to basic client publish manager.

OnCompleted()

This API is used to publish the operation completed event. The API is as follows:

- **OnCompleted(uintptr_t requestId, uasdk::Status_t result, IntrusivePtr_t <**

  PublishResponse_t > const & response)

This API publishes the operation completed response from the server and returns the operation status as result. The *requestId* parameter indicates the input asynchronous callback identifier.

RepublishResponseHandler_t

Different RepublishResponseHandler APIs are mentioned below:

- **Initialise()**
- **OnCompleted()**

Initialise()

This API is used to initialize the republish response handler. The API is as follows:

- **Initialise(BasicClientPublishManager_t * a_parent, const IntegerId_t&**

  subscriptionId)

This API returns the operation status as result. The *parent* parameter specifies the reference to basic client publish manager.

OnCompleted()

This API is used to publish the operation completed event. The API is as follows:

- **OnCompleted(uintptr_t requestId, uasdk::Status_t result, IntrusivePtr_t <**

  RepublishResponse_t > const & response)

This API republishes the operation completed response from the server and returns the operation status as result. The *requestId* parameter indicates the input asynchronous callback identifier.

RecoverableClientSession_t

The Client part of SDK provides a recoverable client session class which enables the automatic recovery for a broken session. It has some important APIs which are listed below:

- **StartHeartBeating()**
- **StopHeartBeating()**
- **Initialise()**
- **GetSession()**
- **RecoverableSessionListener()**

StartHeartBeating()

This API is used to start the heart beating to detect if the session is active on the server, if the session is closed in the server, the recovery is triggered automatically.

- **Status_t StartHeartBeating(IntrusivePtr_t<ITimerProvider_t> timeProvider_,**

  uint32_t delayInMs)

It returns the status which indicates if the operation is successful or not. The *delayInMs* tells how fast the heart beat should be.

StopHeartBeating()

This API is used to stop the heart beat.

- **void StopHeartBeating()**

Initialise()

It is an initializer of the recoverable session.

- **Status_t Initialise(IntrusivePtr_t<IClientSession_t> session, uint32_t**

  timesToRetryReestablishment, uint32_t

  intervalOfReestablishmentInSecs, uint32_t

  maxMonitoredItemsPerRequest)

In this API method, the *session* refers to the original session which is wrapped by the recoverable session.

The *timesToRetryReestablishment* is the maximun number of attempts to reestablish till the reestablishment is successful.

The *intervalOfReestablishmentInSecs* is the interval denoting in seconds between two restablishing attempts.

The *maxMonitoredItemsPerRequest* refers to how many monitored items are allowed in a monitored items creation request while recreating monitored items.

GetSession()

This API returns the wrapped session.

RecoverableSessionListener()

The recoverable session can set and fetch the recoverable session listener.

RecoverableClientSession_t::IRecoverableClientSessionListener_t

The IRecoverableClientSessionListener_t works with RecoverableClientSession_t like ISessionListener working with IClientSession_t. If RecoverableClientSession_t is used to enable the session automatic recovery feature, the UA client application listens to many callbacks which occur during recovery via registering the instance of interface IRecoverableClientSessionListener_t to the recoverable client session. Refer to the the **API Reference Manual** for detailed documentation on each callback method.

IClientConfiguration_t

This API provides a set of getters and setters. With this interface, user developers can get the default settings or configure some basic features of their UA client applications. The detailed configuration items are like the server configuration. Refer to the **Runtime Configuration APIs** section for more information.

IClientCore_t

This is an interface which provides the core functionalities like creating sessions, security settings and discovery service set for a UA client application.

Different IClientCore APIs are mentioned below:

- **Create()**
- **CreateConfiguration()**
- **CreateAllocator()**
- **Initialise()**

- **Start()**
- **Stop()**
- **Shutdown()**
- **GetConfiguration()**
- **SecurityValidateApplicationInstanceCertificate()**
- **SecurityRefreshCertificateDirectoryStores()**
- **SecurityCreateCertificateDirectoryStores()**
- **FindServers()**
- **BeginFindServers()**
- **FindServersOnNetwork()**
- **BeginFindServersOnNetwork()**
- **GetEndPoints()**
- **BeginGetEndpoints()**
- **RegisterServer()**
- **BeginRegisterServer()**
- **EstablishAnonymousSession()**
- **BeginEstablishAnonymousSession()**
- **EstablishUsernamePasswordSession()**
- **BeginEstablishUsernamePasswordSession()**
- **EstablishUserCertificateSession()**
- **BeginEstablishUserCertificateSession()**

Create()

This API is used to create the instance of the client core. The API is as follows:

- **Create(Status_t & result)**

This API returns an intrusive reference counting boost-style smart pointer to the IClientCore_t. The caller must test the pointer before dereferencing it. Parameter *result* specifies the output that returns the server creation status.

CreateConfiguration()

This API is used to create the client configuration. The API is as follows:

- **CreateConfiguration(Status_t & result)**

This API returns an intrusive reference counting boost-style smart pointer to the IClientConfiguration_t. The caller must test the pointer before dereferencing it. Parameter *result* specifies the output that returns the server creation status.

CreateAllocator()

This API is used to create an allocator on the heap. The API is as follows:

- **CreateAllocator(uint8_t * buffer, uint32_t bufferLengthInBytes, uint32_t**

maxAllocationSizeInBytes)

This API returns the operation status as result.

The *buffer* parameter is the input pointer to the buffer.

The parameter *bufferLengthInBytes* indicates the length of the buffer in bytes.

MaxAllocationSizeInBytes parameter is the input that specifies the maximum allocation size in bytes.

Initialise()

This API is used to initialize the client. The API is as follows:

- **Initialise(IntrusivePtr_t < IClientConfiguration_t > configuration, IntrusivePtr_t < IApplication_t > application, IntrusivePtr_t < IThreadPool_t > threadpool, IntrusivePtr_t < ITimerProvider_t > timerProvider, IntrusivePtr_t < IOperationCompleteListener_t > onComplete)**

This API returns the initialize operation status as result.

The *configuration* parameter is the input and is used for configuring the server.

The parameter *threadpool* describes the object to support multi-threading.

TimerProvider parameter indicates the timer object to be used by the server.

On successful completion of the initialize operation, parameter *onComplete* is used to callback OperationComplete of IOperationCompleteListener_t.

Start()

This API is used to start the client operation. The API is as follows:

- **Start(IntrusivePtr_t < IOperationCompleteListener_t > onComplete)**

This API returns the operation status as result. On successful completion of the start operation, parameter *onComplete* is used to callback OperationComplete of IOperationCompleteListener_t.

Stop()

This API is used to stop the client operation. The API is as follows:

- **Stop(IntrusivePtr_t < IOperationCompleteListener_t > onComplete)**

This API returns the stop operation status as result. On successful completion of the stop operation, parameter *onComplete* is used to callback OperationComplete of IOperationCompleteListener_t.

Shutdown()

This API is used to shutdown the client operation. The API is as follows:

- **Shutdown(void)**

This API returns the shutdown operation status as result.

GetConfiguration()

This API is used to get the client configuration. The API is as follows:

- **GetConfiguration(Status_t & result)**

This API returns an intrusive reference counting boost-style smart pointer to the IServerConfiguration_t. The caller must test the pointer before dereferencing it. Parameter *result* specifies the output which returns the server configuration status.

SecurityValidateApplicationInstanceCertificate()

This API is used to validate the application instance certificate based on the security policy. The API is as follows:

- **SecurityValidateApplicationInstanceCertificate(SecurityPolicy_t securityPolicy)**

This API returns the operation status as result. The *policy* parameter is the input and is used for the security policy configuring.

SecurityCreateSelfSignedApplicationInstanceCertificate()

This API is used to create the self-signed application instance certificate. The API is as follows:

- **SecurityCreateSelfSignedApplicationInstanceCertificate(IntrusivePtr_t < const CertificateGenerationParameters_t > parameters, IntrusivePtr_t < IOperationCompleteListener_t > onComplete)**

This API returns the operation status as result.

The *parameters* are the input which specifies the certificate.

On successful completion of the operation, parameter *onComplete* is used to callback OperationComplete of IOperationCompleteListener_t.

SecurityRefreshCertificateDirectoryStores()

This API is used to refresh the certificate directory stores. The API is as follows:

- **SecurityRefreshCertificateDirectoryStores(UA_CertificateGroup_t group, IntrusivePtr_t < IOperationCompleteListener_t > onComplete)**

This API returns the operation status as result.

The *group* parameter is the input and specifies the certificate group.

On successful completion of the start operation, parameter *onComplete* is used to callback OperationComplete of IOperationCompleteListener_t.

SecurityCreateCertificateDirectoryStores()

This API is used to create the certificate directory stores. The API is as follows:

- **SecurityCreateCertificateDirectoryStores(IntrusivePtr_t < IOperationCompleteListener_t > onComplete)**

This API returns the operation status as result. On successful completion of the start operation, parameter *onComplete* is used to callback OperationComplete of IOperationCompleteListener_t.

FindServers()

This API is used to find the UA servers. The API is as follows:

- **FindServers (const IntrusivePtr_t < FindServersRequest_t > request,**

  IntrusivePtr_t<IClientChannel_t> useChannel, IntrusivePtr_t < FindServersResponse_t > & response)

This API returns the operation status as result. The *request* parameter is the input which indicates the reference to the FindServers request. The parameter *response* describes the result of FindServers response.

In case of inbound connection channel parameter which was created as part of reverse connect should be passed for discovery serverices. The useChannel parameter has to be provided in that case.

BeginFindServers()

This API is used to begin the asynchronous find server operation. The API is as follows:

- **BeginFindServers(const IntrusivePtr_t < FindServersRequest_t > request,**

  IntrusivePtr_t<IClientChannel_t> useChannel, IntrusivePtr_t < IFindServersComplete_t > & onComplete)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier. Parameter *request* describes the input which is the reference to FindServers request.

OnComplete parameter indicates the reference to the callback function when the operation is complete.

In case of inbound connection channel parameter which was created as part of reverse connect should be passed for discovery serverices. The useChannel parameter has to be provided in that case.

GetEndPoints()
This API is used to get the available endpoints. The API is as follows:

- **GetEndpoints(const IntrusivePtr_t < GetEndPointsRequest_t > request,**

  IntrusivePtr_t<IClientChannel_t> useChannel, IntrusivePtr_t < GetEndpointsResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the GetEndPoints request.

The parameter *response* describes the result of GetEndpoints response.

In case of inbound connection channel parameter which was created as part of reverse connect should be passed for discovery serverices. The useChannel parameter has to be provided in that case.

BeginGetEndpoints()

This API is used to begin the asynchronous get endpoints operation. The API is as follows:

- **BeginGetEndpoints(const IntrusivePtr_t < GetEndpointsRequest_t > request,**

  IntrusivePtr_t<IClientChannel_t> useChannel,
  IntrusivePtr_t < IGetEndpointsComplete_t > &onComplete)

This API returns the operation status as result.

Parameter *request* describes the input which is the reference to the FindServers request.

OnComplete parameter indicates the reference to the callback function when the operation is complete.

FindServersOnNetwork()

This API is used to find the servers on network. The API is as follows:

- **FindServersOnNetwork(const IntrusivePtr_t<const EndpointDescription_t> discoveryServerEndpoint, const IntrusivePtr_t<FindServersOnNetworkRequest_t> request, IntrusivePtr_t<IClientChannel_t> useChannel, IntrusivePtr_t<FindServersOnNetworkResponse_t>& response)**

This API returns the operation status as result.

The required parameters are endpoint description, request to finding servers on network, client channel. The parameter *response* describes the result of GetEndpoints response.

In case of inbound connection channel parameter which was created as part of reverse connect should be passed for discovery serverices. The useChannel parameter has to be provided in that case.

BeginFindServersOnNetwork()

This API is used to begin the find severs o network. The API is as follows:

- **BeginFindServersOnNetwork(const IntrusivePtr_t<const EndpointDescription_t> discoveryServerEndpoint, const IntrusivePtr_t<FindServersOnNetworkRequest_t> request, IntrusivePtr_t<IClientChannel_t> useChannel, IntrusivePtr_t<IFindServersOnNetworkComplete_t> onComplete)**

This API returns the operation status as result.

The required parameters are endpoint description, request to finding servers on network, client channel.

OnComplete parameter indicates the reference to the callback function when the operation is complete.

In case of inbound connection channel parameter which was created as part of reverse connect should be passed for discovery serverices. The useChannel parameter has to be provided in that case.

RegisterServer()

This API is used to register the server. The API is as follows:

- **RegisterServer(const String_t & discoveryServerUrl, const IntrusivePtr_t**

  <RegisterServerRequest_t > request, IntrusivePtr_t

  <RegisterServerResponse_t > & response)

This API returns the operation status as result.

Parameter *discoveryServerUrl* describes the input which is the value of discovery server URL.

Parameter *request* describes the input which is the reference to the register server request.

The parameter *response* describes the result of the register server response.

BeginRegisterServer()

This API is used to begin the asynchronous register server operation. The API is as follows:

- **BeginRegisterServer(const String_t & discoveryServerUrl, const IntrusivePtr_t <**

  RegisterServerRequest_t > request, IntrusivePtr_t <

  IRegisterServerComplete_t > onComplete)

This API returns the operation status as result.

Parameter *discoveryServerUrl* describes the input which is the value of the discovery server URL.

Parameter *request* describes the input which is the reference to the pointer.

OnComplete parameter indicates the reference to the callback function when the operation is complete.

EstablishAnonymousSession()

This API is used to create the anonymous session. The API is as follows:

- **EstablishAnonymousSession(IntrusivePtr_t < const EndpointDescription_t > serverEndpoint, IntrusivePtr_t < const ClientSessionParameters_t >optionalParameters, IntrusivePtr_t < IClientSession_t & session)**

This API returns the operation status as result.

Parameter *serverEndpoint* describes the input which is the endpoint of the server.

OptionalParameters describes the inputs which are the session parameters for example, timeout etc.

Session parameter indicates to the output anonymous session created.

BeginEstablishAnonymousSession()

This API is used to begin the asynchronous anonymous session created. The API is as follows:

- **BeginEstablishAnonymousSession(uint32_t requestId, IntrusivePtr_t < const EndpointDescription_t > serverEndpoint, IntrusivePtr_t < const ClientSessionParameters_t > optionalParameters, IntrusivePtr_t < IEstablishSessionComplete_t > onComplete)**

This API returns the operation status as result.

Parameter *requestId* describes the input which is the asynchronous callback identifier.

ServerEndpoint parameter describes the endpoint of the server.

OptionalParameters describes the inputs which are the session parameters for example, timeout etc.

OnComplete is the input parameter which indicates the reference to the callback function when the operation is complete.

EstablishUsernamePasswordSession()

This API is used to create the username/password session. The API is as follows:

- **EstablishUsernamePasswordSession(IntrusivePtr_t < const EndpointDescription_t >serverEndpoint, IntrusivePtr_t < const ClientSessionParameters_t > optionalParameters, const String_t & username const String_t & password, IntrusivePtr_t < IClientSession_t > & session)**

This API returns the operation status as result.

Parameter *serverEndpoint* describes the input which is the endpoint of the server.

OptionalParameters describes the inputs which are the session parameters for example, timeout etc.

Username is the input parameter which refers to the username of the session.

Password is the input parameter which refers to the password of the session.

The output *session* parameter describes the created username/password of the session.

BeginEstablishUsernamePasswordSession()

This API is used to begin the asynchronous username/password session creation. The API is as follows:

- **BeginEstablishUsernamePasswordSession(uint32_t requestId, IntrusivePtr_t < const EndpointDescription_t > serverEndpoint, IntrusivePtr_t < const**

ClientSessionParameters_t > optionalParameters, const String_t & username, const String_t & password IntrusivePtr_t < IEstablishSessionComplete_t > onComplete)

This API returns the operation status as result.

Parameter *requestId* describes the input which is the asynchronous callback identifier.

ServerEndpoint parameter indicates the endpoint of the server. *OptionalParameters* describes the inputs which are the session parameters for example, timeout etc.

Username is the input parameter which refers to the username of the session.

Password is the input parameter which refers to the password of the session.

The output *onComplete* parameter represents the reference to the callback function when the operation is complete.

EstablishUserCertificateSession()

This API is used to create the user certificate session operation. The API is as follows:

- **EstablishUserCertificateSession(IntrusivePtr_t < const EndpointDescription_t > serverEndpoint, IntrusivePtr_t < const ClientSessionParameters_t > optionalParameters, IntrusivePtr_t < const IClientUserCertificate_t > userCertificate, IntrusivePtr_t < IClientSession_t > & session)**

This API returns the operation status as result.

Parameter *serverEndpoint* describes the input which is the endpoint of the server.

OptionalParameters describes the inputs which are the session parameters for example, timeout etc.

UserCertificate is the input parameter which refers to the user certificate.

The output *session* parameter describes the created anonymous session.

BeginEstablishUserCertificateSession()

This API is used to begin the asynchronous user certificate session creation. The API is as follows:

- **BeginEstablishUserCertificateSession(uint32_t requestId, IntrusivePtr_t < const EndpointDescription_t > serverEndpoint, IntrusivePtr_t < const ClientSessionParameters_t > optionalParameters, IntrusivePtr_t < const IClientUserCertificate_t > userCertificate, IntrusivePtr_t < IEstablishSessionComplete_t > & onComplete)**

This API returns the operation status as result.

Parameter *requestId* describes the input which is the asynchronous callback identifier.

ServerEndpoint parameter indicates the endpoint of the server.

OptionalParameters describes the inputs which are the session parameters for example, timeout etc.

UserCertificate is the input parameter which refers to the user certificate of the session.

The input *onComplete* parameter represents the reference to the callback function when the operation is complete.

IClientMonitoredItem_t

This is an interface used by the end users for adding the customized data and passing them to the client monitored item object.

Different IClientMonitoredItem APIs are mentioned below:

- **Subscription()**
- **MonitoredItemId()**
- **Item()**
- **MonitoringMode()**
- **ClientHandle()**
- **SamplingInterval()**
- **Filter()**
- **QueueSize()**
- **DiscardOldest()**
- **FilterResult()**
- **ContextHandle()**
- **CreateMonItemModifyRequest()**

Subscription()

These APIs are used to get the owning subscription. The APIs are as follows:

- **IClientSubscription_t & Subscription()**

- **IClientSubscription_t const & Subscription()const**

These functions return the owning subscription as result.

MonitoredItemId()

This API is used to get the server assigned monitored item ID. The API is as follows:

- **IntegerId_t const & MonitoredItemId()const**

This function returns the value of the server assigned monitored item ID as result.

Item()

This API is used to get the assigned monitored item from the create request. This function is read-only and cannot be modified after creation. The API is as follows:

- **ReadValueId_t const & Item()const**

This function returns the value of the assigned monitored item as result.

MonitoringMode()

This API is used to get the assigned monitored mode from the create request. This function is read-only and cannot be modified after creation. The API is as follows:

- **MonitoringMode_t const & MonitoringMode()const**

This function returns the value of the assigned monitored mode as result.

ClientHandle()

This API is used to get the client handle from the create request. This function is read-only, set and used by the SDK. The API is as follows:

- **UInt32_t const & ClientHandle()const**

This function returns the value of the assigned client handle as result.

SamplingInterval()

This API is used to get the sampling interval from the server. This function is read-only and is used to modify IClientSubscription_t::ModifyMonitoredItems. The API is as follows:

- **Duration_t const & SamplingInterval()const**

This function returns the value of the assigned sampling interval as result.

Filter()

This API is used to get the filter parameters from the create request. This function is read-only and is used to modify IClientSubscription_t::ModifyMonitoredItems. The API is as follows:

- **ExtensibleParameter_t const & Filter()const**

This function returns the value of the assigned filter as result.

QueueSize()

This API is used to get the queue size from the server. This function is read-only and is used to modify IClientSubscription_t::ModifyMonitoredItems. The API is as follows:

- **Counter_t const & QueueSize()const**

This function returns the value of the assigned queue size as result.

DiscardOldest()

This API is used to get the set value of the **discard oldest from the create request**. This function is read-only and is used to modify IClientSubscription_t::ModifyMonitoredItems. The API is as follows:

- **Boolean_t const & DiscardOldest()const**

This function returns the Boolean value of the discard oldest state as result

FilterResult()

This API is used to get the filter result from the server. This function is read-only. The API is as follows:

- **ExtensibleParameter_t const & FilterResult()const**

This function returns the value of the filtered item as result.

ContextHandle()

These APIs are used to get and set the handle for the context. These functions are available for the user-defined data which can be derived from the interface and used as desired. These functions are never used by SDK. The APIs are as follows:

- **IntrusivePtr_t < IClientMonitoredItemContext_t > ContextHandle()const**

- **Status_t ContextHandle(IntrusivePtr_t < IClientMonitoredItemContext_t > const &)**

These functions return the intrusive pointer to Context Handle and the operation status as result.

CreateMonItemModifyRequest()

This API is used to create and initialize a modify request with the current values set on the item. This function is useful for populating ModifyMonitoredItemRequest_t. The API is as follows:

CreateMonItemModifyRequest(Status_t & status)const

This function returns the intrusive pointer to the monitored item modify request as result.

Note:

Do not modify the clientHandle member as this is used by the SDK

IClientSession_t

This API is an instance class that represents a connection between UA client application and UA server application and provides the service sets of the UA client.

Different IClientSession APIs are mentioned below:

- **SessionId()**
- **ClientSessionId()**
- **SessionState()**
- **Channel()**
- **SessionListener()**
- **BeginActivate()**
- **Activate()**
- **BeginClose()**
- **Close()**
- **BeginCancel()**
- **Cancel()**
- **BeginBrowse()**
- **Browse()**
- **BeginBrowseNext()**
- **BrowseNext()**
- **BeginTranslateBrowsePathsToNodeIds()**
- **TranslateBrowsePathsToNodeIds()**
- **BeginRegisterNodes()**
- **RegisterNodes()**
- **BeginUnRegisterNodes()**
- **UnRegisterNodes()**
- **Read()**
- **BeginRead()**
- **BeginWrite()**
- **Write()**
- **BeginCall()**
- **Call()**

- **FindSubscription()**
- **Subscriptions()**
- **SubscriptionCount()**
- **HistoryReadAtTime()**
- **BeginHistoryReadAtTime()**
- **HistoryReadRaw()**
- **BeginHistoryReadRaw()**
- **HistoryReadModified()**
- **BeginHistoryReadModified()**
- **HistoryReadProcessed()**
- **BeginHistoryReadProcessed()**
- **HistoryReadEvent()**
- **BeginHistoryReadEvent()**
- **HistoryDeleteAtTime()**
- **BeginHistoryDeleteAtTime()**
- **HistoryDeleteRawModified()**
- **BeginHistoryDeleteRawModified()**
- **HistoryDeleteEvent()**
- **BeginHistoryDeleteEvent()**
- **HistoryUpdateData()**
- **BeginHistoryUpdateData()**
- **HistoryUpdateEvent()**
- **BeginHistoryUpdateEvent()**
- **HistoryUpdateStructureData()**
- **BeginHistoryUpdateStructureData()**
- **BeginCreateSubscription()**
- **CreateSubscription()**
- **BeginTransferSubscription()**
- **TransferSubscriptions()**
- **BeginDeleteSubscriptions()**
- **DeleteSubscriptions()**
- **BeginSetPublishingMode()**
- **SetPublishingMode()**
- **BeginPublish()**
- **Publish()**
- **BeginRepublish()**
- **Republish()**

SessionId()

This API is used to get the session ID. The API is as follows:

- **SessionId(void)**

This API returns the session ID as result. This function provides result in an intrusive reference counting boost-style smart pointer to the String_t. The caller must test the pointer before dereferencing it.

ClientSessionId()

This API returns the client assigned session ID.

- **uint32_t ClientSessionId() const**

SessionState()
This API returns the session state.

- **State_t SessionState(void) const**

Channel()

This API is used to get the underlying channel. The API is as follows:

- **Channel(void)const**

This API returns the underlying channel as result. This function results in an intrusive reference counting boost-style smart pointer to the IClientChannel_t. The caller must test the pointer before dereferencing it.

SessionListener()

This API is used to get the session listener. The API is as follows:

- **SessionListener(void)const**

- **SessionListener(IntrusivePtr_t < IClientSession_t::ISessionListener_t > list)**

This API returns the session listener as result.

This function results in an intrusive reference counting boost-style smart pointer to the ISessionListener_t. The caller must test the pointer before dereferencing it.

BeginActivate()

This API is used to activate the session with the new channel. The API is as follows:

- **BeginActivate(uintptr_t requestId, IntrusivePtr_t < IClientChannel_t > channel,**

IntrusivePtr_t < IEstablishSessionComplete_t > onComplete)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The parameter *Channel* indicates the reference to the new channel.

The parameter onComplete describes the reference to the callback function when the operation is complete.

Activate()

This API is used to activate the session with the new channel. The API is as follows:

- **Activate(IntrusivePtr_t < IClientChannel_t > channel)**

This API returns the operation status as result. The *channel* parameter indicates the reference to the new channel.

BeginClose()

This API is used to begin the asynchronous close operation. The API is as follows:

- **BeginClose(uintptr_t requestId, bool deleteSubscriptions, bool closeChannel, IntrusivePtr_t**

  < ICloseSessionComplete_t > onComplete)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The deleteSubscriptions parameter is the input which is used to delete the subscriptions.

The parameter *closeChannel* is used to close the underlying channel.

The parameter onComplete describes the reference to the callback function when the operation is complete.

Close()

This API is used to close the session and underlying channel. The API is as follows:

- **Close(bool deleteSubscriptions)**

- **Close(bool deleteSubscriptions, bool closeChannel)**

The *deleteSubscriptions* parameter is used to delete the subscriptions.

The *closeChannel* parameter is used to close the underlying channel.

BeginCancel()

This API is used to begin the asynchronous cancel operation. The API is as follows:

- **BeginCancel(uintptr_t requestId, IntrusivePtr_t < CancelRequest_t > const & request, IntrusivePtr_t < ICancelComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the cancel request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

Cancel()

This API is used to cancel the session operation. The API is as follows:

- **Cancel(IntrusivePtr_t < CancelRequest_t > const & request, IntrusivePtr_t < CancelResponse_t > & response)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the cancel request.

The parameter *response* describes the result of the cancel response.

BeginBrowse()

This API is used to begin the asynchronous browse operation. The API is as follows:

- **BeginBrowse(uintptr_t requestId, IntrusivePtr_t < BrowseRequest_t > const & request,IntrusivePtr_t < IBrowseComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the browse request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

Browse()

This API is used to browse the operation. The API is as follows:

- **Browse(IntrusivePtr_t < BrowseRequest_t > const & request, IntrusivePtr_t <**

  BrowseResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the browse request.

The parameter *response* describes the result of the browse response.

BeginBrowseNext()

This API is used to begin the asynchronous browse next operation. The API is as follows:

- **BeginBrowseNext(uintptr_t requestId, IntrusivePtr_t < BrowseNextRequest_t >**

  const & request, IntrusivePtr_t < IBrowseNextComplete_t >

  const & onComplete)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the browse request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

BrowseNext()

This API is used to cancel the operation. The API is as follows:

- **BrowseNext(IntrusivePtr_t < BrowseNextRequest_t > const & request,**

  IntrusivePtr_t < BrowseNextResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the cancel request.

The parameter *response* describes the result of the cancel response.

BeginTranslateBrowsePathsToNodeIds()

This API is used to begin the asynchronous translate browse paths to the node IDs operation. The API is as follows:

- **BeginTranslateBrowsePathsToNodeIds(uintptr_t requestId, IntrusivePtr_t <**

  TranslateBrowsePathsRequest_t > const &

  request, IntrusivePtr_t <

  ITranslateBrowsePathsToNodeIdsComplete_t

  > const & onComplete)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the translateBrowsePaths request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

TranslateBrowsePathsToNodeIds()

This API is used to translate given paths to the node ID operation. The API is as follows:

- **TranslateBrowsePathsToNodeIds(IntrusivePtr_t <**

TranslateBrowsePathsRequest_t > const &

request, IntrusivePtr_t <

TranslateBrowsePathsResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the translateBrowsePath request.

The parameter *response* describes the result of the translateBrowsePath response.

BeginRegisterNodes()

This API is used to begin the asynchronous register the node operation. The API is as follows:

- **BeginRegisterNodes(uintptr_t requestId, IntrusivePtr_t <**

  RegisterNodesRequest_t > const & request,

  IntrusivePtr_t < IRegisterNodesComplete_t > const &

  onComplete)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the registerNodes request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

RegisterNodes()

This API is used to register the node operation. The API is as follows:

- **RegisterNodes(IntrusivePtr_t < RegisterNodesRequest_t > const & request,**

  IntrusivePtr_t < RegisterNodesResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the registerNodes request.

The parameter *response* describes the result of the registerNodes response.

BeginUnRegisterNodes()

This API is used to begin the asynchronous unregister the node operation. The API is as follows:

- **BeginUnRegisterNodes(uintptr_t requestId, IntrusivePtr_t < UnregisterNodesRequest_t > const & request, IntrusivePtr_t < IUnregisterNodesComplete_t > const & completeCallback)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the unRegisterNodes request.

The parameter *completeCallback* describes the reference to the callback function when the operation is complete.

UnRegisterNodes()

This API is used to unregister the node operation. The API is as follows:

- **UnRegisterNodes(IntrusivePtr_t < UnRegisterNodesRequest_t > const & request, IntrusivePtr_t < UnRegisterNodesResponse_t > & response)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the unRegisterNodes request.

The parameter *response* describes the result of the cancel response.

Read()

This API is used to read the operation and to set the attribute service. The API is as follows:

- **Read(const IntrusivePtr_t < ReadRequest_t > request, IntrusivePtr_t < ReadResponse_t > & response)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the read request.

The parameter *response* describes the result of the read response.

BeginRead()

This API is used to begin the asynchronous read operation. The API is as follows:

- **BeginRead(uintptr_t requestId, const IntrusivePtr_t < ReadRequest_t > request,**

  IntrusivePtr_t < IReadComplete_t > completeCallback)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the read request.

The parameter *completeCallback* describes the reference to the callback function when the operation is complete.

BeginWrite()

This API is used to begin the asynchronous write operation. The API is as follows:

- **BeginWrite(uintptr_t requestId, IntrusivePtr_t < WriteRequest_t > const &**

  request, IntrusivePtr_t < IWriteComplete_t > const &

  completeCallback)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the write request.

The parameter *completeCallback* describes the reference to the callback function when the operation is complete.

Write()

This API is used to write the operation. The API is as follows:

- **Write(IntrusivePtr_t < WriteRequest_t > const & request, IntrusivePtr_t <**

  WriteResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the write request.

The parameter *response* describes the result of the write response.

BeginCall()

This API is used to begin the asynchronous call operation and to set the method service operation. The API is as follows:

- **BeginCall(uintptr_t requestId,IntrusivePtr_t < CallRequest_t > const & request,**

  IntrusivePtr_t < ICallComplete_t > const & completeCallback)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the call request.

The parameter *completeCallback* describes the reference to the callback function when the operation is complete.

Call()

This API is used for method call operation. The API is as follows:

- **Call(IntrusivePtr_t < CallRequest_t > const & request, IntrusivePtr_t <**

  CallResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the call request.

The parameter *response* describes the result of the call response.

FindSubscription()

This API is used to find the subscription by subscription ID. The API is as follows:

- **FindSubscription(IntegerId_t const & subscriptionId, Status_t & status)**

This API returns the client subscription status as result.

This function results in an intrusive reference counting boost-style smart pointer to the IClientSubscription_t. The caller must test the pointer before dereferencing it.

Subscriptions()

This API is used to get all the subscriptions. The API is as follows:

- **Subscriptions(Status_t & status)const**

This API returns the iterator list of subscription as result. The parameter *status* indicates the result of the operation. This function results in an intrusive reference counting boost-style smart pointer to the IClientSubscription_t. The caller must test the pointer before dereferencing it.

SubscriptionCount()

This API is used to get the total number of subscriptions. The API is as follows:

- **SubscriptionCount( )const**

This API returns the total count of subscriptions as result.

BeginCreateSubscription()

This API is used to begin the asynchronous create subscription. The API is as follows:

- **BeginCreateSubscription(uintptr_t requestId, IntrusivePtr_t <**

  CreateSubscriptionRequest_t > const & request,

  IntrusivePtr_t < IEstablishSubscriptionComplete_t >

  const & onComplete)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the create subscription request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

CreateSubscription()

This API is used to create the subscription. The API is as follows:

- **CreateSubscription(IntrusivePtr_t < CreateSubscriptionRequest_t > const & request, IntrusivePtr_t < IClientSubscription_t > & subscription)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the create subscription request.

BeginTransferSubscription()

This API is used to begin the asynchronous transfer subscription. The API is as follows:

- **BeginTransferSubscription(uintptr_t requestId, IntrusivePtr_t < TransferSubscriptionsRequest_t > const & request, IntrusivePtr_t < ITransferSubscriptionsComplete_t > & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the transfer subscription request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

TransferSubscriptions()

This API is used to transfer the subscriptions. The API is as follows:

- **TransferSubscriptions(IntrusivePtr_t < TransferSubscriptionsRequest_t > const &**

request, IntrusivePtr_t < TransferSubscriptionsResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the transfer subscription request.

Parameter *response* describes the result of the transfer response.

BeginDeleteSubscriptions()

This API is used to begin the asynchronous delete subscription. The API is as follows:

- **BeginDeleteSubscriptions(uintptr_t requestId, IntrusivePtr_t < DeleteSubscriptionsRequest_t > const & request, IntrusivePtr_t < IDeleteSubscriptionsComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the delete subscription request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

DeleteSubscriptions()

This API is used to delete the subscriptions. The API is as follows:

- **DeleteSubscriptions(IntrusivePtr_t < DeleteSubscriptionsRequest_t > const & request, IntrusivePtr_t < DeleteSubscriptionsResponse_t > & response)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the delete subscription request. Parameter *response* describes the result of the delete response.

BeginSetPublishingMode()

This API is used to begin the asynchronous setting publishing mode. The API is as follows:

- **BeginSetPublishingMode(uintptr_t requestId, IntrusivePtr_t < SetPublishingModeRequest_t > const & request, IntrusivePtr_t < ISetPublishingModeComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the publishing mode request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

SetPublishingMode()

This API is used to set the publishing mode. The API is as follows:

- **SetPublishingMode(IntrusivePtr_t < SetPublishingModeRequest_t > const & request, IntrusivePtr_t < SetPublishingModeResponse_t > & response)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the publishing mode request.

Parameter *response* describes the result of the publishing mode response.

BeginPublish()

This API is used to begin the asynchronous publishing. The API is as follows:

- **BeginPublish(uintptr_t requestId, IntrusivePtr_t < PublishRequest_t > const & request, IntrusivePtr_t < IPublishComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the publishing request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

Publish()

This API is used to publish the operation. The API is as follows:

- **Publish(IntrusivePtr_t < PublishRequest_t > const & request, IntrusivePtr_t < PublishResponse_t > & response)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the publishing request.

Parameter *response* describes the result of the publish response.

BeginRepublish()

This API is used to begin the asynchronous republishing. The API is as follows:

- **BeginRepublish(uintptr_t requestId, IntrusivePtr_t < RepublishRequest_t > const & request, IntrusivePtr_t < IRepublishComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the republishing request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

Republish()

This API is used to republish the operation. The API is as follows:

- **Republish(IntrusivePtr_t < RepublishRequest_t > const & request,**

  IntrusivePtr_t < RepublishResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the republishing request.

Parameter *response* describes the result of the republish response.

IClientSession_t::ISessionListener_t

This interface could be registered into IClientSession_t to listen to the session's state changes.

ChannelInterrupted ()

The callback API is called when session's channel is interrupted (tcp connection is broken).

ChannelReconnected()

The callback API is called when session's channel is reconnected.

ChannelClosed()

The callback API is called when session's channel is closed.

SessionClosed()

The callback API is called when session is closed.

IClientChannel_t

A client channel is a communication channel that ensures the confidentiality and integrity of all messages exchanged with the Server. A client channel is owned either by a client session or shared by multiple client sessions.

ChannelId()

This API method returns the channel ID.

SecurityPolicy()

This API method returns the security policy of the channel.

SecurityMode()

This API method returns the security mode of the channel.

ChannelListener()

Sets or gets the channel state changing listener via this API method.

Close()

This API method is applied to close a channel.

ChannelState()

This API method returns the current state of the channel.

IClientChannel_t::IChannelListener_t

This interface could be registered into IClientChannel_t to listen to the channel's state changes.

ChannelInterrupted ()

The *callback* API is called when channel is interrupted (tcp connection is broken).

ChannelReconnected()

The *callback* API is called when channel is reconnected.

ChannelClosed()
The *callback* API is called when channel is closed.

IClientSubscription_t

This API is an instance class that represents a subscription which monitors a set of monitored items for notifications. It provides the service sets of managing the subscriptions and monitored items.

Different *IClientSubscription* APIs are mentioned below:

- **SubscriptionId()**
- **ClientSubscriptionId()**
- **Session()**
- **FindMonitoredItemById()**
- **FindMonitoredItemByClientHandle()**
- **MonitoredItems()**
- **MonitoredItemCount()**
- **CreateModifySubscriptionRequest()**
- **PublishingInterval()**
- **LifetimeCount()**
- **MaxKeepAliveCount()**
- **MaxNotificationsPerPublish()**
- **Priority()**
- **PublishingEnabled()**
- **BeginModify()**
- **Modify()**
- **BeginCreateMonitoredItems()**
- **CreateMonitoredItems()**
- **BeginModifyMonitoredItems()**
- **ModifyMonitoredItems()**
- **SetMonitoringMode()**
- **BeginSetMonitoringMode()**
- **SetTriggering()**
- **BeginSetTriggering()**
- **DeleteMonitoredItems()**
- **BeginDeleteMonitoredItems()**

SubscriptionId()

This API is used to get the server assigned subscription ID. The API is as follows:

- **IntegerId_t SubscriptionId( )const**

This API returns the value of the server assigned subscription ID as result.

ClientSubscriptionId()

This API returns the client assigned subscription ID:

- **uint32_t ClientSubscriptionId() const**

Session()

These APIs are used to get the owning session. The APIs are as follows:

- **IClientSession_t & Session()**

- **IClientSession_t const & Session()const**

These functions return the owning subscription as result.

FindMonitoredItemById()

This API is used to find a monitored item by its ID. The API is as follows:

- **FindMonitoredItemById(IntegerId_t const & monitoredItemId, Status_t &status)**

This API returns the found monitored item as result.

The *monitoredItemId* parameter indicates the input monitored item assigned by the server. The output parameter *status* describes the operation status.

FindMonitoredItemByClientHandle()

This API is used to find a monitored item by its handle. The API is as follows:

- **FindMonitoredItemByClientHandle(UInt32_t const & clientHandle, Status_t & status)**

This API returns the found monitored item as result.

The *clientHandle* parameter indicates the ID of a monitored item which is assigned by the client. The output parameter *status* describes the operation status.

MonitoredItems()

This API is used to get an iterator of the monitored item collection in the subscription. The API is as follows:

- **MonitoredItems(Status_t & status)const**

This API returns the iterator to access all the managed monitored items as result. The output parameter *status* describes the operation status.

MonitoredItemCount()

This API is used to get a count of the monitored items collection in the subscription. The API is as follows:

- **MonitoredItemsCount( )const**

This API returns the count as result.

CreateModifySubscriptionRequest()

This API is used to initialize a modify request with the current value and the subscription ID. The API is as follows:

- **CreateModifySubscriptionRequest(Status_t & status)const**

This API returns the operation status as result. The output parameter *status* describes the operation status.

PublishingInterval()

This API is used to get the publishing interval. The API is as follows:

- **PublishingInterval(Status_t & status)const**

This API returns the publishing interval status as result. The output parameter *status* returns the operation status.

LifetimeCount()

This API is used to get the lifetime count. The API is as follows:

- **LifetimeCount(Status_t & status)const**

This API returns the lifetime count status as result. The output parameter *status* returns the operation status.

MaxKeepAliveCount()

This API is used to get the maximum keep alive count. The API is as follows:

- **MaxKeepAliveCount(Status_t & status)const**

This API returns the maximum keep alive count status as result. The output parameter *status* returns the operation status.

MaxNotificationsPerPublish()

This API is used to get the maximum notifications per publish. The API is as follows:

- **MaxNotificationsPerPublish(Status_t & status)const**

This API returns the maximum notifications per publish status as result. The output parameter *status* returns the operation status.

Priority()

This API is used to get the priority. The API is as follows:

- **Priority(Status_t & status)const**

This API returns the priority status as result. The output parameter *status* returns the operation status.

PublishingEnabled()

This API is used to get the value of the publishing. To set the value use SetPublishingMode. The API is as follows:

- **PublishingEnabled(Status_t & status)const**

This API returns the value of publishing enabled state as result. The output parameter *status* returns the operation status.

BeginModify()

This API is used to begin the asynchronous modify subscription service call operation. The API is as follows:

- **BeginModify(uintptr_t requestId, IntrusivePtr_t < ModifySubscriptionRequest_t >**

  const & request, IntrusivePtr_t < IModifySubscriptionComplete_t >

  const & onComplete)

This API returns the operation status as result. The *requestId* parameter indicates the input asynchronous callback identifier. The *request* parameter is the input which indicates the modify subscription request. The parameter *onComplete* describes the reference to the callback function when the operation is complete.

Modify()

This API is used to synchronously modify the subscription service call operation. The API is as follows:

- **Modify(IntrusivePtr_t < ModifySubscriptionRequest_t > const & request,**

  IntrusivePtr_t < ModifySubscriptionResponse_t > & response)

This API returns the operation status as result.

The *request* parameter is the input which indicates the modify subscription request.

Parameter *response* describes the server response.

BeginCreateMonitoredItems()

This API is used to begin the asynchronous create monitor items. The API is as follows:

- **BeginCreateMonitoredItems(uintptr_t requestId, IntrusivePtr_t <**

  CreateMonitoredItemsRequest_t > const & request,

  IntrusivePtr_t <

  ICreateClientMonitoredItemsComplete_t > const &

  onComplete)

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to create monitored items request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

CreateMonitoredItems()

This API is used to create the monitored items. The API is as follows:

- **CreateMonitoredItems(IntrusivePtr_t < CreateMonitoredItemsRequest_t > const & request, IntrusivePtr_t < ArrayRef_t < IntrusivePtr_t < MonitoredItemOperationResult_t > > > & items)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to create monitored items request.

The output parameter *items* describe the created monitored items.

BeginModifyMonitoredItems()

This API is used to begin the asynchronous modification of the monitored items. The API is as follows:

- **BeginModifyMonitoredItems(uintptr_t requestId, IntrusivePtr_t < ModifyMonitoredItemsRequest_t > const & request, IntrusivePtr_t < IModifyClientMonitoredItemsComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the modified monitored items request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

The response is returned to the caller and changes are reflected in the corresponding IClientMonitoredItem_t instances before the method returns the output.

ModifyMonitoredItems()

This API is used to modify the monitored items. The API is as follows:

- **ModifyMonitoredItems(IntrusivePtr_t < ModifyMonitoredItemsRequest_t > const & request, IntrusivePtr_t < ArrayRef_t < IntrusivePtr_t < MonitoredItemOperationResult_t > > > & items)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the modified monitored items request.

The output parameter *items* describe the modified monitored items.

The response is returned to the caller and changes are reflected in the corresponding IClientMonitoredItem_t instances before the method returns the output.

SetMonitoringMode()

This API is used to set the monitoring node. The API is as follows:

- **SetMonitoringMode(IntrusivePtr_t < SetMonitoringModeRequest_t > const & request, IntrusivePtr_t < ArrayRef_t < IntrusivePtr_t < MonitoredItemOperationResult_t > > > & items)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the monitored items request.

The output parameter *items* describe the array of the set monitored items operation.

BeginSetMonitoringMode()

This API is used to begin the asynchronous setup of the monitoring mode. The API is as follows:

- **BeginSetMonitoringMode(uintptr_t requestId, IntrusivePtr_t < SetMonitoringModeRequest_t > const & request, IntrusivePtr_t < ISetMonitoringModeComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the set monitoring mode request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

SetTriggering()

This API is used to set the triggering operation. The API is as follows:

- **SetTriggering(IntrusivePtr_t < SetTriggeringRequest_t > const & request, IntrusivePtr_t < SetTriggeringResponse_t > & response)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the set triggering request.

Parameter *response* describes the result of the set triggering response.

BeginSetTriggering()

This API is used to begin the asynchronous setup of triggering. The API is as follows:

- **BeginSetTriggering(uintptr_t requestId, IntrusivePtr_t < SetTriggeringRequest_t > const & request, IntrusivePtr_t < ISetTriggeringComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the set triggering request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

DeleteMonitoredItems()

This API is used to set the delete monitored items operation. The API is as follows:

- **DeleteMonitoredItems(IntrusivePtr_t < DeleteMonitoredItemsRequest_t > const & request, IntrusivePtr_t < DeleteMonitoredItemsResponse_t > & response)**

This API returns the operation status as result.

The *request* parameter is the input which indicates the reference to the delete monitored items request.

Parameter *response* describes the result of the delete monitored item response.

BeginDeleteMonitoredItems()

This API is used to begin the asynchronous delete the monitored items. The API is as follows:

- **BeginDeleteMonitoredItems(uintptr_t requestId, IntrusivePtr_t < DeleteMonitoredItemsRequest_t > const & request, IntrusivePtr_t < IDeleteMonitoredItemsComplete_t > const & onComplete)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The *request* parameter is the input which indicates the reference to the delete monitored items request.

The parameter *onComplete* describes the reference to the callback function when the operation is complete.

IClientUserCertificate_t

This is an interface which provides support to the user certificate based authentication.

Different IClientUserCertificate APIs are mentioned below:

- **GetCertificateDER()**
- **SignData()**

GetCertificateDER()

This API is used to get the available DER encoded certificate. The API is as follows:

- **GetCertificateDER(ByteString_t & certificateDER)const**

This API returns the operation status as result. The output parameter *certificateDER* describes the certificate in DER format.

SignData()

This API is used to request signing of the provided data. The API is as follows:

- **SignData(UA_Digest_t digestType, const ByteString_t & dataToSign, ByteString_t & signature)const**

This API returns the operation status as result.

The *digestType* parameter indicates the digest type to be used.

The parameter *dataToSign* describe the date to be signed. The *Signature* parameter represents the signature.

IModifyClientMonitoredItemsComplete_t

This is a callback interface used to notify the client application that the modification is completed.

The IModifyClientMonitoredItemsComplete API is mentioned below:

OnCompleted()

This API is used to modify the monitored items completed event notified. The API is as follows:

- **OnCompleted(uintptr_t requestId, Status_t status, IntrusivePtr_t < ArrayRef_t < IntrusivePtr_t < MonitoredItemOperationResult_t > > > const & items)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The parameter *items* describe the intrusive pointer to the array of modified item results.

ISetMonitoringModeComplete_t

This is a callback interface used to notify the client application that the setting monitoring mode is completed.

The ISetMonitoringModeComplete API is mentioned below:

OnCompleted()

This API is used to set the monitoring mode completed event notified. The API is as follows:

- **OnCompleted(uintptr_t requestId, Status_t status, IntrusivePtr_t < ArrayRef_t < IntrusivePtr_t < MonitoredItemOperationResult_t > > > const & items)**

This API returns the operation status as result.

The *requestId* parameter indicates the input asynchronous callback identifier.

The parameter *items* describe the intrusive pointer to the array of monitored item operation results.

ITcpConnectionInitiator_t

This is a callback interface used to open a TCP connection.

The ITcpConnectionInitiator API is mentioned below:

Connect()

This API is used to establish the connection. The API is as follows:

- **Connect(const String_t & endpointUrl, IntrusivePtr_t < ITcpConnectionCore_t > &**

connection)

This API returns the operation status as result.

The *endpointUrl* parameter indicates the input which is the endpoint URL to connect. The output parameter *connection* returns the reference to the established connection pointer.

MonitoredItemOperationResult_t

This class encapsulates the result from the operations of creating, modifying and setting the monitored items.

Different MonitoredItemOperationResult APIs are mentioned below:

- **Initialise()**
- **Status()**
- **Item()**

Initialise()

This API is used to initialize the modifiable item. The API is as follows:

- **Initialise(Status_t const & status, IntrusivePtr_t < IClientMonitoredItem_t > &**

  const & item)

This API returns the operation status as result. The *item* parameter indicates the input which is to be modified.

Status()

This API is used to get the recent status. The API is as follows:

- **Status( )const**

This API returns the operation status as result.

Item()

This API is used to get the monitored item. The API is as follows:

- **Item( )const**

This API returns the monitored item status as result.

AggregateType_t

This helper class is used to get the node id of a aggregate function type.

- **static IntrusivePtr_t<NodeId_t> GetAggerateTypeNodeId(AggregateType_t::Enum_t type)**

This API returns the AggregateType Node Id by passing in an aggregate type.

Helper Classes for Reading and Updating Historical Data&Event

The client part of UA SDK provides different helper classes to wrap the orginal history read and update requests and responses by specializing the extensible parameter to different concrete data structures for the different read and update funtionalities.

HistoryReadProcessedRequest_t

This wrapper class is for specializing the general history read request for the ReadProcessed functionality.

HistoryReadProcessedResponse_t

This wrapper class is for specializing the general history read response for the ReadProcessed functionality.

HistoryReadRawRequest_t

This wrapper class is for specializing the general history read request for the ReadRaw functionality.

HistoryReadRawResponse_t

This wrapper class is for specializing the general history read response for the ReadRaw functionality.

HistoryReadModifiedRequest_t

This wrapper class is for specializing the general history read request for the ReadModified functionality.

HistoryReadModifiedResponse_t

This wrapper class is for specializing the general history read rsponse for the ReadModified functionality.

HistoryReadAtTimeRequest_t

This wrapper class is for specializing the general history read request for the ReadAtTime functionality.

HistoryReadAtTimeResponse_t

This wrapper class is for specializing the general history read response for the ReadAtTime functionality.

HistoryReadEventRequest_t

This wrapper class is for specializing the general history read request for the ReadEvent functionality.

HistoryReadEventResponse_t

This wrapper class is for specializing the general history read response for the ReadEvent functionality.

HistoryDeleteAtTimeRequest_t

This wrapper class is for specializing the general history update request for the DeleteAtTime functionality.

HistoryDeleteEventRequest_t

This wrapper class is for specializing the general history update request for the DeleteEvent functionality.

HistoryDeleteRawModifiedRequest_t

This wrapper class is for specializing the general history update request for the DeleteRawModified functionality.

HistoryUpdateDataRequest_t

This wrapper class is for specializing the general history update request for the UpdateData functionality.

HistoryUpdateEventRequest_t

This wrapper class is for specializing the general history update request for the UpdateEvent functionality.

HistoryUpdateStructureDataRequest_t

This wrapper class is for specializing the general history update request for the UpdateStructureData functionality.

PubSub

The main classes associated with PubSub are described below. For detailed documentation please refer to the reference manual. For details of the various settings and configuration options please refer to the OPC UA specification at:

https://reference.opcfoundation.org/v104/Core/docs/Part14/

PublishSubscribeFactory_t

This class allows the creation of IPublishSubscribe objects.

CreatePlatformConfiguration ()

Create a platform configuration object that defines various runtime configuration options specific to how the SDK executes PubSub.

CreateAddressSpace()

Create an address space object.

CreateXmlNodesetIO()

Create an object to facilitate input/output of XML nodesets to/from an address space.

CreatePublishSubscribe()

This set of methods creates an IPublishSubscribe object.

IPublishSubscribe_t

This class is the root class for active publishers and subscribers and contains all child components such as connections, groups, etc.

There are numerous methods the highlights of which are as follows:

GetConnections()

Get the list of PubSubConnections.

PublishedDataSets()

Get the root DataSetFolder that contains the data sets that are published in this subscription.

Status()

Get the state machine for this object.

Diagnostics()

Get the diagnostics.

CreateModel()

Expose the configuration model in the address space. Also exposes the models of all child components.

RefreshModel()

Refresh the configuration model. Also refreshes the models of all child components.

RemoveModel()

Remove the configuration model from the address space. Also removes the models of all child components.

Enable()

Enable the component.

Disable()

Disable the component.

State()

Get the current state of the component state machine.

GetConfiguration()

Get a copy of the configuration.

AddConnection()

Add a new PubSubConnection.

RemoveConnection()

Remove a PubSubConnection.

DataSetFolder_t

This class represents a DataSetFolder. A DataSetFolder contains PublishedDataSets and optionally other DataSetFolders.

There are numerous methods the highlights of which are as follows:

GetPublishedDataSets()

Get the PublishedDataSets contained within the folder.

GetPublishedDataItems()

Get the PublishedDataItems contained within the folder. These are a subset of the PublishedDataSets.

GetPublishedEvents()

Get the PublishedEvents contained within the folder. These are a subset of the PublishedDataSets.

GetDataSetFolders()

Get the sub folders contained within this DataSetFolder.

CreateModel()

Expose the configuration model in the address space. Also exposes the models of all child components.

RefreshModel()

Refresh the configuration model. Also refreshes the models of all child components.

RemoveModel()

Remove the configuration model from the address space. Also removes the models of all child components.

AddPublishedDataItems()

Add an additional PublishedDataSet of DataItems.

AddPublishedEvents()

Add an additional PublishedDataSet of events.

RemovePublishedDataSet()

Remove a PublishedDataSet.

AddDataSetFolder()

Add a DataSetFolder.

RemoveDataSetFolder()

Remove a DataSetFolder.

Create()

Create an instance of this class.

PublishedDataSet_t

This class represents a PublishedDataSet. A PublishedDataSet contains the DataSetMetaData and the PublishedDataSet source information.

There are numerous methods the highlights of which are as follows:

PublishedDataSetName()

Get the name of the PublishedDataSet.

GetDataSetMetaData()

Get the DataSetMetaData.

SetDataSetMetaData()

Set the DataSetMetaData.

GetDataSetWriters()

Get the DataSetWriters that write this PublishedDataSet.

CreateModel()

Expose the configuration model in the address space. Also exposes the models of all child components.

RefreshModel()

Refresh the configuration model. Also refreshes the models of all child components.

RemoveModel()

Remove the configuration model from the address space. Also removes the models of all child components.

AddDataSetWriter()

Add an additional DataSetWriter.

RemoveDataSetWriter()

Remove a DataSetWriter.

Create()

Create an instance of this class.

IPubSubConnection_t

This class represents a PubSubConnection. A PubSubConnection can contain WriterGroups and ReaderGroups.

There are numerous methods the highlights of which are as follows:

ConnectionName()

Get/set the name of the connection.

PublisherId()

Get/set the PublisherId.

Address()

Get/set the network address the connection connects to.

GetWriterGroups()

Get the WriterGroups that are children of this connection.

GetReaderGroups()

Get the ReaderGroups that are children of this connection.

Status()

Get the state machine for this object.

Diagnostics()

Get the diagnostics.

CreateModel()

Expose the configuration model in the address space. Also exposes the models of all child components.

RefreshModel()

Refresh the configuration model. Also refreshes the models of all child components.

RemoveModel()

Remove the configuration model from the address space. Also removes the models of all child components.

Enable()

Enable the component.

Disable()

Disable the component.

State()

Get the current state of the component state machine.

GetConfiguration()

Get a copy of the configuration.

AddWriterGroup()

Add a new WriterGroup.

AddReaderGroup()

Add a new ReaderGroup.

RemoveGroup()

Remove a Group.

ModifyConnection()

Modify the connection.

IWriterGroup_t

This class represents a WriterGroup. A WriterGroup can contain DataSetWriters.

There are numerous methods the highlights of which are as follows:

WriterGroupId()

Get the WriterGroupId.

PublishingInterval()

Get the publishing interval for this group.

KeepAliveTime()

Get the KeepAliveTime.

GetDataSetWriters()

Get the DataSetWriters that are children of this group.

Status()

Get the state machine for this object.

Diagnostics()

Get the diagnostics.

CreateModel()

Expose the configuration model in the address space. Also exposes the models of all child components.

RefreshModel()

Refresh the configuration model. Also refreshes the models of all child components.

RemoveModel()

Remove the configuration model from the address space. Also removes the models of all child components.

Enable()

Enable the component.

Disable()

Disable the component.

State()

Get the current state of the component state machine.

GetConfiguration()

Get a copy of the configuration.

AddDataSetWriter()

Add a new DataSetWriter.

RemoveDataSetWriter()

Remove a DataSetWriter.

ModifyWriterGroup()

Modify the group.

IReaderGroup_t

This class represents a ReaderGroup. A ReaderGroup can contain DataSetReaders.

There are numerous methods the highlights of which are as follows:

GetDataSetReaders()

Get the DataSetReaders that are children of this group.

Status()

Get the state machine for this object.

Diagnostics()

Get the diagnostics.

CreateModel()

Expose the configuration model in the address space. Also exposes the models of all child components.

RefreshModel()

Refresh the configuration model. Also refreshes the models of all child components.

RemoveModel()

Remove the configuration model from the address space. Also removes the models of all child components.

Enable()

Enable the component.

Disable()

Disable the component.

State()

Get the current state of the component state machine.

GetConfiguration()

Get a copy of the configuration.

AddDataSetReader()

Add a new DataSetReader.

RemoveDataSetReader()

Remove a DataSetReader.

ModifyReaderGroup()

Modify the group.

IDataSetWriter_t

This class represents a DataSetWriter.

There are numerous methods the highlights of which are as follows:

DataSetWriterId()

Get the DataSetWriterId.

DataSetFieldContentMask()

Get the DataSetFieldContentMask.

DataSetName()

Get the DataSet name.

DataSetWriterName()

Get the DataSetWriter name.

KeyFrameCount()

Get the KeyFrameCount.

Status()

Get the state machine for this object.

Diagnostics()

Get the diagnostics.

PublishedDataSet()

Get the PublishedDataSet that this DataSetWriter writes.

CreateModel()

Expose the configuration model in the address space. Also exposes the models of all child components.

RefreshModel()

Refresh the configuration model. Also refreshes the models of all child components.

RemoveModel()

Remove the configuration model from the address space. Also removes the models of all child components.

Enable()

Enable the component.

Disable()

Disable the component.

State()

Get the current state of the component state machine.

GetConfiguration()

Get a copy of the configuration.

ModifyDataSetWriter()

Modify DataSetWriter.

IDataSetReader_t

This class represents a DataSetReader.

There are numerous methods the highlights of which are as follows:

DataSetWriterId()

Get the DataSetWriterId.

WriterGroupId()

Get the WriterGroupId.

DataSetFieldContentMask()

Get the DataSetFieldContentMask.

DataSetReaderName()

Get the DataSetReaderName.

DataSetMetaData()

Get the DataSetMetaData for the DataSet read by this reader.

MessageReceiveTimeout()

Get the MessageReceiveTimeout.

KeyFrameCount()

Get the KeyFrameCount.

Status()

Get the state machine for this object.

Diagnostics()

Get the diagnostics.

SubscribedDataSet()

Get the SubscribedDataSet that this DataSetReader reads.

CreateModel()

Expose the configuration model in the address space. Also exposes the models of all child components.

RefreshModel()

Refresh the configuration model. Also refreshes the models of all child components.

RemoveModel()

Remove the configuration model from the address space. Also removes the models of all child components.

Enable()

Enable the component.

Disable()

Disable the component.

State()

Get the current state of the component state machine.

GetConfiguration()

Get a copy of the configuration.

ModifyDataSetReader()

Modify DataSetReader.

Appendix

Appendix A: Third-Party Licenses including Open Source Software

Sl No.	Library Name	Version	Remarks
1.	Boost	1.70.0 (12th April 2019)	• This library is used to get the feature of Intrusive pointer in C++98. • This can be found in path <Path to SDK_Source>\ interface\common\portable\smart_pointers\
2.	TinyXml2	8.0.0 (2nd March 2020)	• This Library is used to parse the nodeset XML file to create address space. • It can be found in path: <path to SDK_Source>\interface\common\platform_specific\tinyxml2\ • Slight modification is done in library for memory optimization • This library will be used if **UASDK_INCLUDE_XML_SUPPORT** and **UASDK_USE_TINYXML2** build macros are set.
3	LibXml2	2.9.10(30th Oct. 2019)	• This library is not a part of the SDK. • If **UASDK_INCLUDE_XML_SUPPORT** and **UASDK_USE_LIBXML2** build macros are set then, this library needed to build the SDK. • This Library is used to parse the nodeset XML file to create address space.
4	OpenSSL	1.0.2u, 1.1.0l and 1.1.1h (22nd Sept. 2020)	• This library is used to support security. • This library is not a part of the SDK. • If **UASDK_INCLUDE_SECURITY** and **UASDK_USE_OPENSSL** build macros are set then, this library is needed to build the SDK.
5	SQLite	3.21.0	• This Library is used to historize the data • This Library is not a part of the SDK • If **UASDK_INCLUDE_HISTORY** is set and examples provided by the packages are used, then this library needed to build the server application and to historize the data
6	mBedTLS	2.16.00 (21st December 2018)	• This library is used to support security. • This library is not a part of the SDK. • If **UASDK_INCLUDE_SECURITY** and **UASDK_USE_MBEDTLS** build macros are set then, this library is needed to build the SDK.
7	lwIP	2.0.0	• This library is used to get the TCP/IP related functionality for server/client. • If OPC UA application is created for some controller where lwIP is used to have the TCP/IP functionality, then **UASDK_INCLUDE_SERVER_LWIP_RAW** build macro need to set and, SDK needs lwIP library to build
10	POSIX	NA	• This library is used to have declaration of POSIX directory browsing functions and types for Win32

Sl No.	Library Name	Version	Remarks
11	STM32Cube V1	NA	This library is used for building SDK for STM32F769 Disc0
12	STM32Cube V2	NA	This library is used for building SDK for STM32F769 Disc0
13	OPC Foundation UA ANSI C Stack	1.04 (19th November 2019)	• This is used for OPC UA status codes. • opcua_status_codes.h is the file and it can be found in the path <Path to SDK>\interface\common\portable\miscellaneous\

Table 5 – Third-Party Components used in SDK

Appendix B: Assessing the amount of RAM required for an embedded UA Server

Introduction

The Matrikon FLEX OPC UA SDK makes extensive use of dynamic memory allocation for memory management. Given the nature of the OPC UA specification, implementing a flexible and scalable OPC UA SDK using only static memory allocation would not be feasible.

The SDK can be configured to perform its dynamic memory allocation using the system heap or an internal allocator. The internal allocator can be selected via the `UASDK_USE_SYSTEM_HEAP` build macro and the single, contiguous block of RAM used by the allocator is provided during initialisation by the application developer.

Estimation technique

In OPC UA, most messages are of variable length and the OPC UA Client has considerable freedom in how it forms messages and interacts with the Server. This means that there is no algorithm that will yield the required amount of memory for a given configuration. In order to identify the amount of memory required it must be calculated experimentally.

The steps are as follows:

1. Configure the allocator with a large amount of memory.
2. Configure the server to limit the behavior of client.
3. Stress the server to the limits of its configured capabilities.
4. Read out the maximum memory used and the recommended memory from the allocator.

Configure the allocator

Configure the allocator by calling `UAServer_t::CreateAllocator()` and providing as much memory as possible. Three arguments must be provided:

- The address of the RAM buffer.
- The size of the buffer in bytes.

- The maximum allocation size in bytes.

The maximum allocation size must be large enough to contain a message of max. message size. Max. message size is configured via the server configuration `IServerConfiguration_t::TCPBinaryMaxMessageSize().`

Configure the server

The server configuration class, `IServerConfiguration_t`, exposes a variety of methods that allow the application developer to place restrictions on what a connected client can do. Many of these restrictions will limit both server RAM and system CPU utilisation.

The most important limits are; `SecureChannelMaxChannels, MaxSessions, MaxSubscriptionsPerSession, MaxNotificationRetransmissionQueueSize, MaxPublishRequestsPerSession, MaxMonitoredItems` and `MaxMonItemQueueSize.`

- `SecureChannelMaxChannels`. The maximum number of channels the server will allow clients to open concurrently. This value must be greater than the maximum number of sessions to ensure that a channel is available for the discovery services when all sessions are in use.
- `MaxSessions.` The maximum number of concurrent sessions. Multiple sessions can be opened within the context of a single channel, but most clients dedicate a channel to each session so in typical usage each session will require a channel.
- `MaxSubscriptionsPerSession.` The maximum number of subscriptions supported per session. Subscriptions result in monitored item creation, monitored item sample queuing and publish response message queuing.
- `MaxNotificationRetransmissionQueueSize.` Each subscription can queue outgoing publish response messages to support the republish service. Set this queue size to 1 to minimise the amount of memory used.
- `MaxPublishRequestsPerSession`. Incoming publish requests are queued by each session. For a given session, this value must be greater than the maximum number of subscriptions but keep it as small as possible to minimise the amount of memory used.
- `MaxMonitoredItems.` The maximum number of monitored items that the server supports in total (sum of all subscriptions).
- `MaxMonItemQueueSize`. The maximum number of data change samples that can be queued within each monitored item. Set this value to 1 to minimise memory usage.

The build macro `UASDK_OPTIMISE_FOR_SPEED` and the macros it controls can also be used to optimise the server for speed or memory utilisation. Setting this macro to 0 will throttle inbound message handling in the server to minimise the amount of messages the server must queue at any given time.

Stress the server to the limits of its configured capabilities

Stress testing can be performed by using an off-the shelf OPC UA Client or a purpose-built test client. A test client is desirable as it gives the developer finer control over Client behaviour. If a test client is not available an off-the-shelf Client such as UA Expert can be used instead. UA Expert has a performance tab that can be used to stress the server by making continuous service calls as well as a data access tab that allows the creation and configuration of subscriptions and monitored items.

A sample test plan could be as follows:

Using a client…

- Open the maximum number of sessions to the server.
- In each session create the maximum number of subscriptions.
- In each subscription create monitored items such that the total sum of items equals the server limit.ss
- In each monitored item configure the max queue size to equal the server limit.
- Configure the monitored item sampling intervals and the subscription publishing interval such that the server will be forced to queue the monitored item samples.
  For example, for a max monitored item queue size of 5 set the monitored item sampling interval to 50ms and the subscription publishing interval to 300ms (assuming a server cyclic rate of 50ms).
- While the subscriptions are running make frequent service calls such as browse, read or write within each session with large numbers of operations.

This combination of actions will heavily load the server resources both in terms of CPU bandwidth and RAM usage.

Read out the memory metrics

The memory metrics can be obtained from the allocator. Get a pointer to the allocator by calling `Allocatable_t::Allocator().` Then call `IAllocator_t::MemoryInfo().` The memory info method returns a variety of information including the total memory committed since initialisation and the total recommended memory. The recommendation takes the maximum number of blocks used and adds at least 50% as a safety margin. This is a recommendation only and its suitability must be assessed by the application developer. If more memory is available than the recommended figure, then more memory can be provided.

The memory info metrics are only meaningful if the allocator has not run out of memory. An out of memory condition can be detected as the SDK will call the global method `UASDK_allocator_out_of_memory().`

A convenient way of watching the metrics is to create a variable in the address space with a value attribute reader writer which can be monitored or read to display the maximum committed memory at a given point in time.