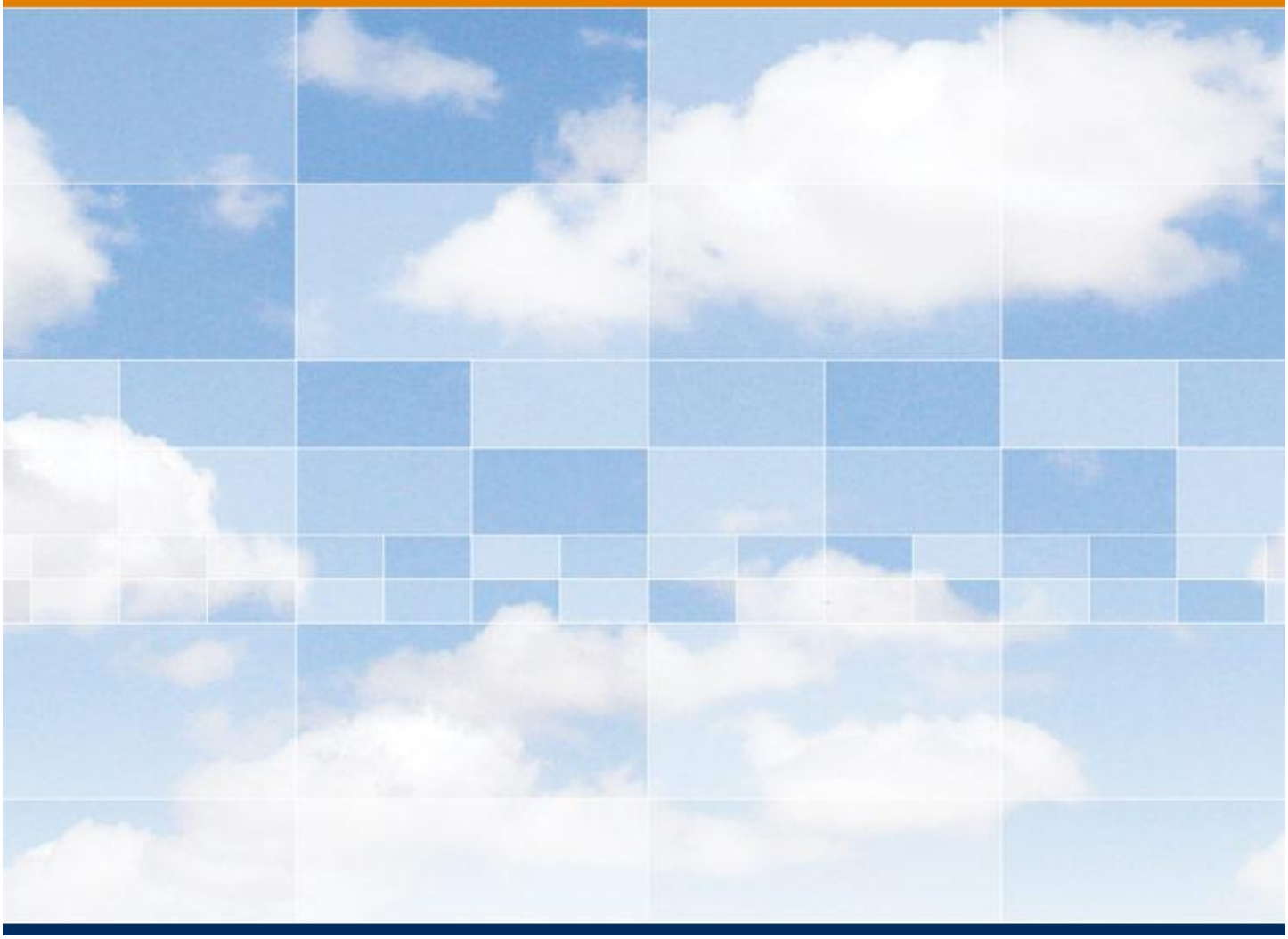
MatrikonOPC



OPC UA Embedded Server SDK

User Manual

OPC UA Embedded Server SDK
User Manual

This manual is a product of Matrikon Inc.

Matrikon Inc.
Suite 1800, 10405 Jasper Avenue
Edmonton, AB T5J 3N4
Canada

Phone: +1.780.448.1010
Fax: +1.780.448.9191
www.matrikonopc.com

Document Revision History:

Date	Document Version	Description	Author
2013-08-15	1.0	Initial revision	LP
2013-08-22	1.01	Amended description of UAServer_Callback_monitored_item() to reflect modified signature.	LP
2014-05-28	2.0	Updated to SDK version 2.0.0	LP
2014-05-31	2.0.1	Added additional security information	LP
2014-09-18	2.2	Added information about MatrixSSL, configuration and tuning information, File Transfer and others	LP
2014-09-29	2.2.1	Added information about dynamic memory based address space, saving rejected certificates.	LP
2014-10-03	2.2.2	Added info about dynamic nodes, IEEE 754, examples environment, entropy. Revised MatrixSSL instructions.	LP
2015-04-09	2.2.3	Added info about multithreading and performance optimisation.	LP
2015-11-30	2.2.4	Changed SDK version number.	LP

SOFTWARE VERSION

Version: 2.1.4

DOCUMENT VERSION

Version: 2.2.4

COPYRIGHT INFORMATION

© Copyright 1997 - 2015, Matrikon Inc. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, translated, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of Matrikon Inc.

CONFIDENTIAL

The information contained herein is confidential and proprietary to Matrikon Inc. It may not be disclosed or transferred, directly or indirectly, to any third party without the explicit written permission of Matrikon Inc.

IMPORTANT NOTICE

Although every endeavor has been made to ensure that the information contained within this document is up to date and accurate, Matrikon cannot be held responsible for any inaccuracy or error in the information contained within this document. Matrikon makes no warranty of any kind with regard to the information contained within this document and Matrikon shall not be liable for any direct, indirect, incidental or consequential damages which may arise in connection with the furnishing, reliance, or use of the information contained within this document.

Specifications and statements as to performance in this document are Matrikon estimates, intended for general guidance. Matrikon reserves the right to change the information contained within this document and any product specification without notice.

Statements in this document are not part of a contract or program product licence insofar as they are incorporated into a contract or licence by express preference. Issue of this document does not entitle the recipient to access or use of the products described, and such access or use shall be subject to separate contracts or licences.

The receiving party shall not disclose, publish, report, communicate, or otherwise transfer any information in this document to any third party, and shall protect all information contained herein from unauthorized disclosure. The receiving party shall permit access to this document only to its employees, agents, subcontractors, and affiliates who reasonably require access to such information contained herein, have been made aware of the confidential nature of this document and have executed a written employment or other confidentiality agreement party to maintain the confidential status of this document.

LICENSE AGREEMENT

This document and the software described in this document are supplied under a license agreement and may only be used in accordance with the terms of that agreement. Matrikon reserves the right to make any improvements and/or changes to product specifications at any time without notice.

TRADEMARK INFORMATION

The following are either trademarks or registered trademarks of their respective organizations:

Matrikon and MatrikonOPC are trademarks or registered trademarks of Matrikon Inc.

OTHER

MatrikonOPC™ is a division of Matrikon™ Inc.

Table of Contents

Introduction 8
Who Should Use This Manual 8
Specification 9
Supported Profiles 9
Supported Transport Protocol....9
Supported Security Facet 9
Limits 9
Address Space 9
Platform Requirements 10
CPU 10
Compiler 10
FLASH and RAM 10
IEEE 754 Floating Point Conformance.... 10
TCP/IP Stack.... 11
Time Synchronisation.... 11
Battery Backed Real Time Clock 11
SNTP (Simple Network Time Protocol) 11
IEEE15888 PTP.... 11
Emulation 11
Runtime Environment 12
SDK Distribution and Examples 13
Distribution Organisation 13
Examples.... 13
Building the SDK 16
Configuration Header 16
UA_MAX_ALIGNMENT.... 16
UA_BIG_ENDIAN 16
UA_CROSS_ENDIAN_DOUBLE 16
UA_INCLUDE_SUBSCRIPTIONS 16
UA_INCLUDE_MONITORED_ITEM_TRIGGERING.... 16
UA_INCLUDE_STANDARD_ADDRESS_SPACE.... 16
UA_OVERLOAD_NEW_AND_DELETE 16
UA_INCLUDE_CPP_STD_LIB_DEPS 17
UA_INCLUDE_APP_DEFINED_NODE_IDS 17
UA_NO_THROW.... 17
UA_STANDARD_NODES_SECTION_ATTRIBUTE_PREFIX.... 17
UA_STANDARD_NODES_SECTION_ATTRIBUTE_POSTFIX.... 17
UA_INCLUDE_SECURITY.... 17
UA_USE_OPENSSL.... 17
UA_USE_MATRIXSSL.... 17
UA_MATRIXSSL_SUPPORT_SECURITY_POLICY_BASIC256 17
UA_MATRIXSSL_SUPPORT_CERTIFICATE_GENERATION 18
APPLICATION_RSA_KEY_LENGTH 18
UA_INCLUDE_ASYNC_OPS.... 18
UA_MUTEX_T 18
ALLOW_USER_TOKEN_POLICY_X509v3_CERTIFICATE_TOKEN_SECURITY_POLICY_NONE 18
UA_OPTIMISE_FOR_SPEED.... 18
UA_DEADBAND_DATA_TYPE_T.... 19
UA_PRODUCT_URI UA_PRODUCT_NAME UA_MANUFACTURER_NAME UA_SOFTWARE_VERSION
UA_BUILD_NUMBER UA_BUILD_DATE.... 19

Considerations 19
SDK Integration 20
API Files 20
General 20
Security 20
API Function Types 20
Basic SDK Operation 20
Date, Time and Standard Library Integration 21
UAServer_Callback_millisecond() 21
UAServer_Callback_datetime() 21
UAServer_Callback_memcpy(), etc. 21
TCP/IP Stack Integration 22
Connection ID 22
New connection 22
Connection closed 22
Receiving data 22
Sending data 22
Locales and Translations 24
UAServer_Callback_translate() 24
UAServer_Callback_translate_internal_text() 24
UAServer_Callback_translate_enumstrings() 24
Configuration and Initialisation 25
UAServer_Initialise() 26
UAServer_Set_URI() 27
UAServer_Add_IP_address() UAServer_Add_hostname() UAServer_Clear_all_IP_addresses()
UAServer_Clear_all_hostnames() 27
Changing IP Addresses and Hostnames 27
UAServer_Generate_certificate() UAServer_Get_certificate_size() UAServer_Read_certificate()
UAServer_Write_certificate() 27
UAServer_Start() 28
UAServer_Stop() UAServer_Get_state() 28
UAServer_Terminate () 28
Populating the Address Space 29
Static versus Dynamic Address Space Generation 29
Reading and Writing Variables 33
Reading Variables 33
Writing Variables 35
Application Defined Node Ids 37
File Transfer 38
Deferred Reads 39
Asynchronous Operation 40
Integrating Security 41
OPC UA Security Concept 41
Certificate Stores 41
Endpoint Configuration 41
Initial Provisioning 42
Trusting All Certificates 42
Self-signed versus CA Issued Certificates 43
Self-signed 43
CA issued 43
Security Error Suppression 43
Saving Rejected Client Certificates 43
Security API 44

UASecurity_Check_application_certificate_exists() 44
UASecurity_Create_self_signed_certificate() 44
UASecurity_Reload_directory_stores_now() 44
UASecurity_Reload_directory_stores_before_each_validation() 44
UASecurity_Create_directory_stores() 44
UASecurity_Set_validation_options() 44
UASecurity_Get_thumbprint() 44
UASecurity_Callback_get_private_key_password() 45
UASecurity_Callback_certificate_added_to_rejected_list() 45
OpenSSL Integration 46
Entropy Generation 46
MatrixSSL Integration 47
Runtime Configuration Settings 51
TCP Binary Stack Layer 51
HELLO_TIMEOUT_IN_MS 51
CONNECTION_TIMEOUT_IN_MS 51
PORT_NUMBER 51
MAX_MESSAGE_SIZE 51
MAX_MESSAGE_CHUNK_SIZE 52
MAX_MESSAGE_CHUNK_COUNT 52
MAX_CONNECTIONS 52
Secure Conversation Stack Layer 52
MIN_REVISED_CHANNEL_LIFETIME_IN_MS MAX_REVISED_CHANNEL_LIFETIME_IN_MS 52
MAX_SECURE_CHANNELS 53
Security 53
Services 54
MIN_REVISED_SESSION_TIMEOUT_IN_MS MAX_REVISED_SESSION_TIMEOUT_IN_MS 54
MAX_SESSIONS 54
MAX_SUBSCRIPTIONS_PER_SESSION 54
MIN_MAX_KEEP_ALIVE_COUNT MAX_MAX_KEEP_ALIVE_COUNT 55
MAX_NOTIFICATION_RETRANSMISSION_QUEUE_SIZE 55
MAX_PUBLISH_REQUESTS_PER_SESSION 55
BROWSE_MAX_REFS_PER_NODE 56
MAX_MONITORED_ITEMS 56
MAX_MON_ITEM_QUEUE_SIZE 56
Informational Arrays 56
NAMESPACE_ARRAY 56
SOFTWARE_CERTIFICATES 57
SERVER_PROFILE_ARRAY 57
LOCALE_ID_ARRAY 57
Server Capabilities 57
MAX_BROWSE_CONTINUATION_POINTS 57
MAX_QUERY_CONTINUATION_POINTS MAX_HISTORY_CONTINUATION_POINTS 57
MAX_ARRAY_LENGTH 57
MAX_STRING_LENGTH 57
Operation Limits 57
ENFORCE_OPERATION_LIMITS 57
MAX_NODES_PER_READ 58
MAX_NODES_PER_HISTORY_READ_DATA MAX_NODES_PER_HISTORY_READ_EVENTS
MAX_NODES_PER_HISTORY_UPDATE_DATA MAX_NODES_PER_HISTORY_UPDATE_EVENTS .. 58
MAX_NODES_PER_WRITE 58
MAX_NODES_PER_METHOD_CALL 58
MAX_NODES_PER_BROWSE 58

MAX_NODES_PER_REGISTER_NODES *58*
MAX_NODES_PER_TRANSLATE_BROWSE_PATHS_TO_NODE_IDS *58*
MAX_NODES_PER_NODE_MANAGEMENT *58*
MAX_MONITORED_ITEMS_PER_CALL *58*
Internals 58
USE_HEAP_RATHER_THAN_RAM_BUFFER *58*
BUFFER BUFFER_LENGTH_IN_BYTES *59*
MAX_ALLOCATION_SIZE_IN_BYTES *59*
ADDRESS_SPACE_HASH_TABLE_NO_OF_BUCKETS *59*
MAX_NO_OF_DEFERRED_RESPONSE_MESSAGES *59*
Diagnostics 60
ENABLE_INFO_LOG *60*
ENABLE_ERROR_LOG *60*
Filters 60
DISABLE_PCT_DEADBAND *60*
DISABLE_PCT_DEADBAND_VECTORS *60*
DISABLE_PCT_DEADBAND_FLOATING_POINT *60*
DISABLE_PCT_DEADBAND_DOUBLE *60*
DISABLE_PCT_DEADBAND_FLOATING_POINT_VECTORS *60*
DISABLE_PCT_DEADBAND_DOUBLE_VECTORS *60*
Intervals 60
INTERVALS_ARRAY INTERVALS_ARRAY_LENGTH *61*
ALLOW_ARBITRARY_INTERVALS *61*
MIN_INTERVAL_IN_MS *61*
MAX_INTERVAL_IN_MS *61*
CYCLIC_RATE_IN_MS *61*
Memory Usage and Performance 62
Performance Issues 62
Memory Utilisation 63
Other Features and Further Information 65
User Groups 65
Views 65
API Reference Manual 65
Compliance Certification 65
What you need to know about OPC UA 68
OPC-UA Server Concept 68
Address Space 69
Nodes, References, Attributes and Properties *69*
Folders *69*
Views *69*
Variables *69*
Variable Types *70*
Sessions and Requests 72
Subscriptions and Monitored items 72

Introduction

The OPC UA Embedded Server SDK is a Software Development Kit for embedded OPC UA Servers that delivers the following benefits:

- A small code and data footprint suitable for deployment on single chip microcontrollers.
- An extremely fast and efficient implementation.
- An easy to use API that allows you to quickly integrate the server into your device.
- A software architecture designed from first principles for robustness and reliability.

Who Should Use This Manual

This document is targeted at skilled embedded software engineers to provide the information they need to successfully integrate the SDK into their product. A basic knowledge of OPC UA is assumed. For readers with no knowledge of OPC UA a brief overview of the standard is provided in the appendix.

This document uses icons to highlight valuable information. Remember these icons and what they mean, as they will assist you throughout the manual.

	This symbol denotes important information that must be acknowledged.
BOLD	Font displayed in this color and style indicates a hyperlink to the applicable/associated information within this document, or if applicable, any external sources.

Specification

Supported Profiles

- Embedded Server Profile
- Data Access Server Facet
- Standard DataChange Subscription Server Facet

Supported Transport Protocol

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

Supported Security Facet

- SecurityPolicy – None
- SecurityPolicy – Basic128Rsa15
- SecurityPolicy – Basic256

Limits

- No limits on number of concurrent sessions, subscriptions, monitored items or address space nodes (subject to available resources)
- 16 user group limit (including the anonymous user)

Address Space

- Supports dynamic creation and deletion of the following node and reference types:
    - Folders
    - Views
    - BaseDataVariables
    - Data Access node types (DataItem, AnalogItem, TwoState and MultiState)
    - Files
    - Hierarchical references

Platform Requirements

CPU

- A 32 bit CPU

Compiler

- A C++ compiler that conforms with ***ISO/IEC 14882***:*1998* (C++98)

Note that while the SDK is implemented in C++ the API is ANSI C and there is no requirement for the main device application to be implemented in C++. Typically device applications are implemented in ANSI C and interact with the SDK via the C API.

FLASH and RAM

Table 1: Minimum Server Footprint

Configuration	Flash (kB)*	RAM (kB)
Nano Embedded Device Server Profile	350	50
Micro Embedded Device Server Profile	400	65
Embedded Server Profile (including security and full address space)	520	100

*Metrics obtained for ARM Thumb2 instruction set (Cortex-M4F), Atollic TrueSTUDIO 4.20, GCC -Os

These are minimum ROM and RAM requirements for the SDK only, excluding the resources required by the device application and the TCP/IP stack. These are minimums and depending on the functional requirements of a particular application in terms of address space contents, number of concurrent connections, monitored items, etc. the actual resources required can be much higher.

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 so these metrics must be viewed in context of the test architecture.

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 512kB Flash and 128kB RAM or 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 1MB Flash and 256kB RAM or an external SRAM IC.

IEEE 754 Floating Point Conformance

The SDK requires a runtime environment that conforms to IEEE 754. Certain compiler options such as GCC "fastmath" can impair this conformance resulting in undefined behavior. If your 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 instead provide a block based API. The SDK can be integrated with any block based API.

For microcontrollers we provide examples integrating with the open source TCP/IP stack LwIP (**http://savannah.nongnu.org/projects/lwip/**) in both raw and socket mode. We recommend 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 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 uIP or the Keil 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 in order to facilitate support of the OPC UA Local Discovery functionality which shall be released as an extension to the standard in the near future.

Time Synchronisation

All OPC UA Servers require the host system to provide them with date and time information for time stamping various server activities. Time can be provided in a number of ways:

Battery Backed Real Time Clock

SNTP (Simple Network Time Protocol)

An SNTP Client application can be integrated into the device. Such clients can either listen for SNTP broadcasts from a time server located on the LAN, or alternatively can request time updates periodically from specific SNTP servers. A number of free NTP server applications are available for download and it is normal practice in Industrial Automation networks to have an NTP Server present to synchronise 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

Where high resolution time synchronisation is required IEEE1588 Precision Time Protocol can be 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 synchronisation to the order of 100ns between devices on a LAN or even WAN (via GPS clock synchronisation).

Emulation

For development purposes the SDK can be placed in time emulation mode whereby it sets its clock based on the time stamps on incoming client requests. This mode of operation is not standard compliant and is not suitable for production but can be very useful during development.

Runtime Environment

The SDK is by default a single threaded implementation with API functions that are called by the device application and application callback functions that are called by the SDK. The implementation is not re-entrant or thread safe and all calls must be made on the same thread. Where the application developer chooses to distribute calls across threads all resource locking is the responsibility of the application developer.

The SDK can be implemented within a continuous loop in a bare metal environment (main function with no OS) or within a single thread of an OS or RTOS.

The application developer can optionally enable an asynchronous mode whereby work items are provided to the application by the SDK via the API. The application can then run these work items on a separate worker thread or threads to reduce Server latency and monitored item sample jitter.

SDK Distribution and Examples

Distribution Organisation

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

Distribution
- Examples
  - Main_Examples_Linux_Eclipse_CDT
  - Linux_MAKEPP_Makefile
  - Win32_VS2008
- HDK
  - C# NTP Server
  - HDK Documentation
  - TrueSTUDIO_bare_metal
  - TrueSTUDIO_external_ram_no_security
  - TrueSTUDIO_external_ram_with_security
  - TrueSTUDIO_internal_ram
- SDK Documentation
  - API Reference
- SDK_Source
  - API
  - configuration templates
  - internals

Examples

The desktop is a highly productive environment to experiment with the SDK and learn how to use the API. Version 2.0.4 of the SDK does not support native 64 bit compilation. For this reason developing on a 64 bit platform such as Linux or Windows requires building the application as a 32 bit application.

The examples directory contains a variety of examples that allow you to develop with the SDK.

Linux Eclipse CDT (Main Examples Environment)

A Linux Eclipse CDT workspace allows for development in the Eclipse IDE on x86 Linux platforms. The Eclipse workspace used to create the example workspace is Luna. This is the main examples environment containing a variety of example projects that work out of the box on x86 Linux. If you wish to run these examples on a different platform you will need to port them manually.

It is recommended to build the SDK as a static library and link this to your application in order to speed up development. The Linux Eclipse CDT examples do not follow this convention in order to simplify the Eclipse workspace and make it as easy to use as possible.

If you wish to use the Linux CDT examples out of the box and do not have a Linux system available follow these steps:

1. Install 32 bit Ubuntu Desktop 14.04 LTS into a Virtualbox or VMWare Virtual Machine.
2. Update the package maanger 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. Download and unzip Eclipse Luna for C/C++ developers (Linux 32-bit).
7. Open the example workspace and work with the included examples.

The included examples are as follows:

Nano Profile Server
A tiny server supporting a minimum amount of functionality as specified by the OPC Foundation Nano Embedded Device Server Profile.

Micro Profile Server
A small server supporting a limited functionality as specified by the OPC Foundation Micro Embedded Device Server Profile.

Embedded Profile with OpenSSL
A small server supporting security as specified by the OPC Foundation Embedded Server Profile.

User Node IDs
A server that uses string, numeric and GUID node ids rather than using the default opaque node ids.

Static Memory Address Space
A server that uses static tables to allocate process data and to look up that data from callbacks.

Dynamic Memory Address Space
A server that uses dynamic memory to allocate process data and to look up that data from callbacks. This is the recommended mode of operation for all but the smallest of servers.

File Transfer
A server that implements the OPC Foundation File Transfer functionality.

OPCF CTT Compliance Test
A server that implements an information model suitable for running the OPC Foundation Compliance Test Tool against. An example CTT configuration file is included. Version 2.0.4 passes the Embedded Server Profile and Data Access Profile test cases with CTT version 1.2.335.239.

Deferred Read
A server that implements the deferred reading of some variables from its data source.

Asynchronous Security
A server that offloads slow security operations to a worker thread to reduce server latency and reduce sample jitter.

Linux MAKEPP Makefile

A MAKEPP makefile based project is provided to facilitate simple makefile based development on x86. To build the SDK library and example project change directory to "Distribution/Examples/Linux_MAKEPP_Makefile/opcua_example" and execute "makepp opcua_example". Both the makefile and Eclipse CDT projects can be used to easily 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 compilation and linker flags that your platform requires.

Visual Studio 2008

A Visual Studio 2008 solution is provided for developing in a Windows environment. The Visual C++ 2008 Example will only build if OpenSSL is installed at C:\OpenSSL-Win32.

Installers can be obtained from **https://slproweb.com/products/Win32OpenSSL.html** at your own risk. In order to build the example with no security follow the following steps:

1. Remove libeay32.lib from the additional dependency list found in the ExamplesServer properties page (Linker->Input).

2. Remove "C:\OpenSSL-Win32\lib" from the additional library directories list found in the ExamplesServer properties page (Linker->General).

3. Set UA_INCLUDE_SECURITY to zero in ua_server_config.h.

HDK

The HDK directory contains documentation and a number of Attolic TrueSTUDIO workspaces that contains example applications for an STM32F4 ARM Cortex-M4F microcontroller. The examples include a bare metal and FreeRTOS based projects using the open source LwIP TCP/IP stack. A simple NTP Broadcast Server is also provided in C# and can be used for time synchronisation of the HDK board to the Client PC during development and test.

SDK Documentation and Source

The SDK documentation folder contains a user manual in PDF format along with an API reference manual in HTML format located at “Distribution/SDK Documentation/API Reference/index.html”. The SDK source folder contains the SDK API, the SDK internals in obfuscated source code format and configuration templates for the two files need to build and configure a server.

Building the SDK

Configuration Header

To build the SDK into a static library the header file ua_server_config.h must be produced. This can be produced from the template, ua_server_config_template.h. The default settings from the template are suitable for many platforms. The configuration macros are:

UA_MAX_ALIGNMENT

The maximum alignment of the CPU. The default value is 8.

UA_BIG_ENDIAN

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

UA_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.

UA_INCLUDE_SUBSCRIPTIONS

Include support for Subscriptions and Monitored Items. The default value is 1 (enables the feature).

UA_INCLUDE_MONITORED_ITEM_TRIGGERING

Include support for Monitored Item Triggering. The default value is 1 (enables the feature).

UA_INCLUDE_STANDARD_ADDRESS_SPACE

The Nano and Micro Embedded Device Server Profiles can optionally include full type information in their Address Space. This information increases the flash footprint by approximately 100kB but is highly recommended for interoperability purposes as many OPC UA Clients expect this information to be present. If interoperability is not a major concern due to the use of a custom client then this type information can be omitted to reduce the flash requirement. The default value is 1 (enables the feature). The Embedded Server Profile requires the standard address space to be present.



Warning. If you build the SDK as a library and link it to the application in a subsequent step you must use the same ua_server_config.h to build your application. This is because the address space text table included in the API is selected via this build macro and it must match the setting used to build the SDK.

UA_OVERLOAD_NEW_AND_DELETE

The SDK is implemented in C++ but has no standard library or static class initialisation dependencies. What this means in practice is that on many platforms, the SDK can be integrated into a “C project” which uses the standard C library and C startup code. The C++ standard library can be omitted and the project startup code does not need to be modified. When integrating the SDK in this manner, some linkers will fail to link the project due to the absence of new and delete functions. Setting this macro to 1 causes new and delete to be overridden in the SDK. The default value is 0.

UA_INCLUDE_CPP_STD_LIB_DEPS

The SDK is implemented in C++ but has no standard library or static class initialisation dependencies. What this means in practice is that on many platforms, the SDK can be integrated into a "C project" which uses the standard C library and C startup code. The C++ standard library can be omitted and the project startup code does not need to be modified. When integrating the SDK in this manner certain C++ standard library functions need to be stubbed out. Setting this macro to 1 causes stubs for C++ to be implemented. Please note that this is not possible with every toolchain. The default value is 0 (disables the feature).

UA_INCLUDE_APP_DEFINED_NODE_IDS

By default the SDK assigns an opaque Node ID to every address space node created by the application. This macro enables the application to specify node id's for the nodes it creates of type string, GUID and numeric. The default value is 0 (disables the feature).

UA_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 a number of places in the SDK even when exceptions are disabled. Other compilers will fail to compile if exception specifications are specified when exceptions are disabled. Set this macro to throw() or an empty macro to suit your compiler. The default value is throw().



Warning. Setting this macro to anything other than throw() on some platforms can result in segment violations or hard faults in out of memory conditions so modify this macro with care.

UA_STANDARD_NODES_SECTION_ATTRIBUTE_PREFIX

UA_STANDARD_NODES_SECTION_ATTRIBUTE_POSTFIX

The standard address space is approximately 100kB of constant data that can be stored in ROM or flash. If you require to specify a specific linker section to store this data use these macros to do so.

UA_INCLUDE_SECURITY

This macro enables full security functionality in the SDK.

UA_USE_OPENSSL

Enables the OpenSSL PKI component interface module.

UA_USE_MATRIXSSL

Enables the MatrixSSL PKI component interface module.

UA_MATRIXSSL_SUPPORT_SECURITY_POLICY_BASIC256

Enables support for SecurityPolicyBasic256 in MatrixSSL. This policy uses algorithms that are not present in the open source version of MatrixSSL, but are present in the commercial version. The macro enables building the SDK with the open source version for test purposes prior to purchasing a commercial MatrixSSL license.

UA_MATRIXSSL_SUPPORT_CERTIFICATE_GENERATION

Enables support for self-signed certificate generation in MatrixSSL. Certificate generation uses algorithms that are not present in the open source version of MatrixSSL, but are present in the commercial version. The macro enables building the SDK with the open source version for test purposes prior to purchasing a commercial MatrixSSL license.

APPLICATION_RSA_KEY_LENGTH

Specifies the key length in bits of the self-signed application instance certificate that can be created when security is included. Default value is 1024 bits (legal values are 1024 and 2048 bits).

UA_INCLUDE_FILE_TRANSFER_METHODS

This macro enables file transfer functionality in the SDK. The default value is 0 (disables the feature)

UA_INCLUDE_READ_DEFERRALS

This macro allows the host application to defer some read callbacks while it obtains the data values from slow data sources. The default value is 0 (disables the feature)

UA_INCLUDE_ASYNC_OPS

This macro enables the asynchronous operations feature. This feature causes the SDK to deliver work items to the application via the API. The application may then run these work items on a separate worker thread or threads for reduced latency and sample jitter. The work items are those that require RSA private key operations to be performed as these can be very slow on a microcontroller platform.



Warning. Enabling this macro will result in some API Callback functions being called on a different thread. The API calls affected are documented in the reference manual. It is critical that these API callback functions are thread safe to avoid undefined behaviour. This is the responsibility of the application developer.

UA_MUTEX_T

When enabling the asynchronous operations feature the SDK requires access to mutexes to protext memory allocation, etc. The platform specific mutex datatype must be declared here so that the SDK can declare mutexes as required.

ALLOW_USER_TOKEN_POLICY_X509v3_CERTIFICATE_TOKEN_SECURITY_POLICY_NONE

When using X.509 certificates for user authentication the OPC UA standard allows for specifying a user token policy of "Security None". This effectively eliminates the signature verification for the user token which is a security hole. By default the SDK will not allow a user token policy for X.509 certificates that does not require signature verification. Enabling this macro overrides this protection and allows the behaviour.

UA_OPTIMISE_FOR_SPEED

This macro configures the SDK to favour speed over RAM usage. A number of optimisations are affected which reduce the time to perform the Browse and TranslateBrowsePathsToNodeIds

services. Message encoding and decoding time is also dramatically reduced. Where the Server is configured with a maximum message size that is much larger than the maximum message chunk size then this macro can dramatically increase memory utilisation.

UA_DEADBAND_DATA_TYPE_T

When the percent deadband filter is enabled in the configuration it converts data values to doubles in order to perform the deadband calculation. On platforms such as the ARM Cortex-M4F double precision floating point can be much slower than single precision due to the absence of a double precision hardware floating point unit. In this case, changing the deadband data type to float, when combined with suitable compiler options can improve percent deadband performance on selected platforms.

UA_PRODUCT_URI
UA_PRODUCT_NAME
UA_MANUFACTURER_NAME
UA_SOFTWARE_VERSION
UA_BUILD_NUMBER
UA_BUILD_DATE

These macros define constant product and manufacturer related info that is exposed in the address space. These macros should be configured correctly prior to releasing your product to market.

Considerations

- The library should be built where possible with separate sections for every function and variable to facilitate stripping of the binary on linkage to reduce code size.
- The library should be built where possible with RTTI and Exceptions disabled.
- The library makes use of limit macros such as UINT32_MAX, etc. which are defined by the inclusion of the macro __STDC_LIMIT_MACROS.

SDK Integration

Where the SDK is built as a static library, the library must be added to an application project along with the API files.

API Files

General

- ua_types.h. This header contains the user defined types required to interact with the library.
- ua_server.h. This header defines the ANSI C functional interface.
- ua_server_config.h. This is the same header used to build the SDK library. It is also required when building the application.
- opcua_display_name_identifiers.c/h. This module/header includes the English browse and display names for all standard nodes in the server Address Space. These strings are exposed as part of the API so that additional translations of these strings can be added if required. These is no obligation to add translations for the standard strings.
- opcua_status_codes.h. The status codes defined by the OPC UA standard that are used throughout the API.
- opcua_cpp_std_lib_stubs.c. See configuration header UA_INCLUDE_CPP_STD_LIB_DEPS above.

Security

- ua_security.h. This is a header file used by the SDK to interact with the PKI interface modules. The application does not need to include this file.
- ua_security_openssl.c. The OpenSSL PKI interface module.
- ua_security_matrixssl.c. The MatrixSSL PKI interface module.

API Function Types

The API contains two types of function, those that are called by the application and those called by the SDK but which must be implemented by the application. All API functions are prefixed with UAServer_ while callback functions are prefixed with UAServer_Callback_.

Basic SDK Operation

The SDK is a simple single threaded implementation. In order to integrate the SDK into an application successfully the following tasks must be performed:

1. Implement all API Callback functions.
2. Call API TCP/IP functions in response to TCP/IP stack activity (new connections, new data, etc.)
3. Call UAServer_Poll() at the defined cyclic rate (see configuration and initialisation).

Date, Time and Standard Library Integration

There are two callback functions relating to date and time.

UAServer_Callback_millisecond()

This callback must return an unsigned 32 bit integer that represents an incrementing counter in milliseconds. Although the counter units are milliseconds the counter does not have to increment with 1 millisecond resolution. Any period <= the server cyclic rate is acceptable.

UAServer_Callback_datetime()



Warning. This callback must return a signed 64 bit integer in Windows FILETIME format. This is the number of 100ns intervals since January 1, 1601. This function can be called with a very high frequency to timestamp various server operations. If a large number of monitored items are being sampled this callback will execute for every sample. For this reason while it is not mandatory to optimise the callback, doing so can dramatically improve performance.

Taking embedded Linux as an example, calling gettimeofday() in UAServer_Callback_datetime () results in a context switch which is a very slow operation. Where large numbers of variables are being monitored this system call can account for close to 50% of the application CPU utilisation. Implementing a Virtual Dynamic Shared Object on Linux, or a similar mechanism on another OS to avoid a context switch in this callback can dramatically improve performance.

UAServer_Callback_memcpy(), etc.

There are many callback functions that provide for integration with standard library calls such as memcpy, memset, etc. This integration is performed via callbacks for two reasons.

1. In some cases the SDK may be successfully linked to a C standard library (rather than C++) and some C++ standard library stubs. This makes integration of the SDK into pre-existing C projects easier on platforms where this is possible and prudent. Not referencing the standard library calls within the body of the SDK makes this possible.
2. Where performance optimisation is critical, an application may wish to implement their own versions of memcpy, etc. to make use of DMA and/or 32 bit operations for maximum speed. Exposing the standard library call hooks makes this possible.

TCP/IP Stack Integration

The TCP/IP interface is the most complex part of the API. The complexity is a result of two requirements, to enable the use of block based as well as socket based TCP/IP stacks and to minimise the copying of data so as to maximise performance on resource constrained hardware.

Example code is provided to map the API to Windows sockets, Berkeley sockets and to the LwIP raw API. If your TCP/IP stack API is significantly different to these examples you will have to perform the integration yourself.

For microcontrollers we provide examples integrating with the open source TCP/IP stack LwIP in both raw and socket mode. We recommend however that customers use a commercial TCP/IP stack in their product.

The integration requirements are outlined below.

Connection ID

Each concurrent TCP/IP connection from a UA Client is uniquely identified in the SDK by a connection id. From the library's perspective this is an opaque identifier (an int64_t) that must be unique for each active connection. Where a socket based API is in use the socket number can be used for the connection_id. Otherwise the remote endpoint can be used (remote IP Address and port number).

New connection

When a client establishes a new TCP connection call UAServer_New_connection() with the new connection_id.

Connection closed

When a client terminates an open TCP/IP connection call UAServer_Connection_closed().

Receiving data

Where new incoming data is available on a connection call UAServer_New_data_received(). The server will repeatedly call UAServer_Callback_data_receive() to read in the new data into its TCP Binary buffer.
Depending on your TCP/IP stack the server may occasionally be unable to receive additional data for processing until it can complete sending previously generated response data. In this case, after calling UAServer_New_data_received() not all of the received data will have been read from the TCP/IP stack. It is the responsibility of the application in combination with the TCP/IP stack to buffer this data until the server is ready to accept it and to adjust receiver windows and throttle network communications as necessary to handle this. In the case of socket based TCP/IP stacks this behaviour is typically performed automatically by the TCP/IP stack and the application has to do nothing. In the case of block based TCP/IP stacks the correct application behaviour is TCP/IP stack specific.

Sending data

After calling UAServer_Poll() or UAServer_New_data_received() the application should call UAServer_Get_data_to_send(). After polling this should be called for all open connections and after receiving new data it should be called for the connection that has just received data. The server

responds by repeatedly calling UAServer_Callback_data_send() to transfer any outgoing data to the TCP/IP stack.

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 are "en" for English and "en-US" for English (USA). A locale table is provided to the server as part of the configuration. The first entry is the default locale.
All responsibility for translating strings based on the locale in use in a particular session is the responsibility of the application and is performed through the implementation of 3 functions; UAServer_Callback_translate(), UAServer_Callback_translate_internal_text() and UAServer_Callback_translate_enumstrings().

UAServer_Callback_translate()

UAServer_Callback_translate() is called when the server wants to get a BrowseName or DisplayName for a node added by the application. The locale in use is indicated by the locale_index. This is the index into the locale table provided during configuration. The text required is indicated by the text_handle. This is the unique display_name_handle provided when the node was created in the server address space. A server implementing a single locale may ignore the locale_index.

UAServer_Callback_translate_internal_text()

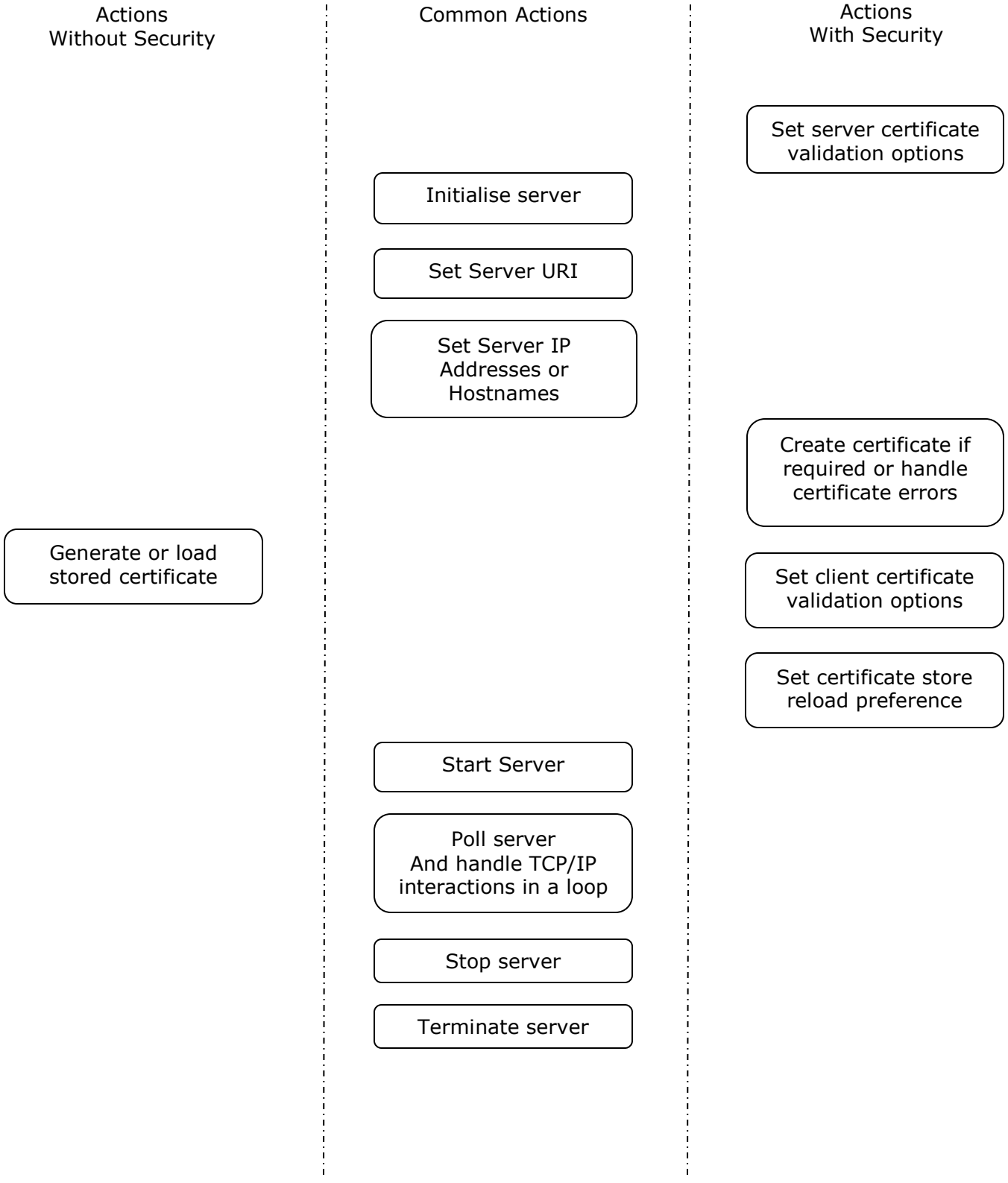
UAServer_Callback_translate_internal_text() is provided to enable the application to provide localized strings for the standard nodes in the address space such as the Server Object, type information, etc. It is expected that very few applications will do so in which case the default implementation of this callback can be used without modification.

UAServer_Callback_translate_enumstrings()

Where the server exposes MultiStateDiscrete DA types in the address space the names of the enumerations must also be translated. UAServer_Callback_translate_enumstrings() provides a means of translating enumerations.

Configuration and Initialisation

The basic steps required to bootstrap the SDK are shown graphically below.



The API functions that must be called in order to initialise and configure the server are outlined below.

UAServer_Initialise()

The UAServer_Initialise() function call allocates internal resources and performs initialisation and basic configuration of the server. The configuration structure passed to the function must remain in scope for the entire lifetime of the application as the server does not copy this structure but repeatedly refers to it via the pointer provided. The configuration structure is described in detail in a later section.



Warning. Where security is not included UAServer_Initialise() will always return OpcUa_Good (zero) when initialisation was successful and the server may be started. When security is included the SDK must validate the security configuration and this may result in the return of some non-zero status codes that are not necessarily terminal. On receipt of one of these error codes the application must decide whether or not it is prudent to continue.

The validation of the security configuration depends on the security validation options. These options may be set by making the call UASecurity_Set_validation_options(). **This is the only SDK API call that can safely be made before calling UAServer_Initialise()**.

The non-terminal error codes that the SDK may return are as follows:

- OpcUa_BadNoValidCertificates. An application instance certificate and private key are not found. The initialisation was successful but a certificate must be created or installed in order to proceed successfully.

- OpcUa_BadSecurityChecksFailed. The application instance certificate failed validation for an undetermined reason (it may be corrupted). The initialisation was successful but if a new certificate is not created or installed the server may not interoperate correctly.

- OpcUa_BadCertificateTimeInvalid. The application instance certificate failed validation due to an invalid time (it may have expired). The initialisation was successful but if a new certificate is not created or installed the server may not interoperate correctly. Also, operating the server with an expired certificate may be against your stated security policy.

- OpcUa_BadCertificateRevoked. The application instance certificate failed validation because it has been revoked or because the CA CRL could not be found offline. The initialisation was successful but if a new certificate is not created or installed the server may not interoperate correctly. Also, operating the server with a revoked certificate may be against your stated security policy.

If you wish to suppress the OpcUa_BadCertificateTimeInvalid and/or OpcUa_BadCertificateRevoked error codes for your application instance certificate, either ignore these error codes or set the appropriate security validation option bits prior to initialisation. Validation option bits are global, they affect Client certificates too, therefore if you require different options for client certificates change the option bit configuration after UAServer_Initialise() by again calling UASecurity_Set_validation_options() with the options specific to Client certificates.

UAServer_Set_URI()

The OPC-UA standard specifies that every server must have a unique id. This id is provided by a Uniform Resource identifier. This URI is passed to the library as a null terminated C string. There are many standard forms of URI such as those based on URN's (Universal Resource names) containing UUID's (Universally Unique Identifiers)
e.g. "urn:uuid:6e8bc430-9c3a-11d9-9669-0800200c9a66"
or URN's based on MAC addresses, e.g. "urn:dev:mac:0024befffe804ff1".

A standard form of URI does not need to be used. It is legal to use any form of unique identifier. By definition URIs must be unique. Where unique URIs are not used and in the absence of security, clients could easily not detect connection to the wrong server where server IP Addresses have changed for example. For further information about URI's see the relevant IETF standards.

UAServer_Add_IP_address()
UAServer_Add_hostname()
UAServer_Clear_all_IP_addresses()
UAServer_Clear_all_hostnames()

The hostnames and/or IP Addresses through which the server is accessible should be configured via these functions.

Changing IP Addresses and Hostnames

Changing the URI and/or IP address(es) and/or hostname(s) affects the contents of the server Application Instance Certificate as well as the Server Endpoints returned by the discovery services. Changing these values at runtime can break an active communication session between a client and the server.

DHCP results in volatile IP Addresses and for this reason DHCP should only be used on systems that support hostname resolution of some kind. When using DHCP the system would add a hostname and not bind the SDK configuration to a volatile IP Address.

The impending Local Discovery functionality when released will support hostname resolution based on Multicast DNS to allow microcontroller based systems use DHCP as an aid to configuration and management.

UAServer_Generate_certificate()
UAServer_Get_certificate_size()
UAServer_Read_certificate()
UAServer_Write_certificate()

Servers supporting no security can optionally provide an Application Instance Certificate to a Client during the discovery and session negotiation phases. While providing no security functionality this function must be called after setting the URI and hostnames to ensure interoperability with older Clients. Certificate generation is a slow process involving RSA signing and can take some time to complete on a microcontroller. An execution time of 300ms on an MCU would not be unusual.

To avoid slow start up the application can store the generated certificate in a file rather than generating it every time. The certificate needs to be regenerated of the IP Address or Hostname of the server device changes. The server URI should never change.

UAServer_Start()

Places the server in the running state and the server will now process incoming requests.

UAServer_Stop()
UAServer_Get_state()

Places the server in the shutdown state. The server may take up to 60 seconds to shutdown if clients are connected.

UAServer_Terminate ()

Terminates the server and releases all internal resources.

Populating the Address Space

In order to make the server do anything useful we need to expose our process data so that Clients can interact with our process. In OPC Classic we create OPC Tags. In OPC UA we create nodes which we expose in the Server Address Space.

Exposing a data point is a two-step process. Firstly we must create the node in the address space. Secondly we must create a reference from a source node to our new node (the target node) so that the node can be discovered by browsing the address space.

In the current release the variables that can be created are BaseDataVariable nodes, Data Access nodes and FileType nodes. These can be exposed by creating hierarchical references from Folder nodes to each variable node. This is analogous to placing the variable in a folder.

The address space contains a standard node called the Objects node. This node is a Folder to which we can add other nodes. It is possible to create a node and add the same node to multiple folders. If you create a node but do not add it to a folder (create a reference) then the node can be read (and possibly written to) but it cannot be discovered by browsing and the Client must have prior knowledge of the Node ID in order to access it. This would be an unusual implementation and normally nodes are always referenced by other nodes and browseable by the Client.

Static versus Dynamic Address Space Generation

When creating a variable in the address the actual OPC UA address space node is allocated and stored internally within the SDK.

The application must create and store:

- The actual process value
- Source timestamp
- Value status code
- The name of the node (the display name).

Regarding the node name, the SDK uses the default locale DisplayName as the BrowseName to simplify the API. If you are not clear on what these terms mean and are creating a server supporting a single language you can simply ignore them.

A number of approaches can be taken to store this data and the correct approach depends on your application and platform requirements. For very small servers the data can be stored statically using tables. For more advanced servers the data can be dynamically allocated using the SDK allocator. *Examples are provided that illustrate both methods*. Unless your application address space is very small we recommend using the SDK allocator to dynamically generate this data.



Warning. If you provide pointers to the SDK as node handles then you must use application defined node ids to ensure that node ids are persistent, i.e. they do not change if you power cycle the server. Even if you are dynamically allocating node data structures using the SDK allocator, it is not necessary to use pointers for node handles, but it can be very convenient to do so. Remember that using application defined node ids uses more memory and slows down your server.

Here is an example of dynamically creating a folder and placing it in the Objects Folder, creating integer and string variables and then placing the variables in the folder.

```
void Add_nodes(void)
{
  UA_Status_t status = 0;

  /* We are using pointers to data structures as node handles. Folder handles are never read
   * or written by the SDK so we don't need a data structure to store anything but we still
   * require a unique node handle. For this reason we can just allocate a byte to get a unique handle.
   * If you ever delete the node from the server you will need to free this byte yourself by
   * calling UAServer_Free.
   */
  uint8_t* my_dummy_byte = UAServer_Malloc(1);
  if (!my_dummy_byte)
  {
    printf("UAServer_Malloc failed\n");
    return;
  }

  //Add Folder
  UA_Folder_t folder;
  UAServer_Init_config_structure_Folder(&folder);
  folder.node_handle =(uint32_t)my_dummy_byte;
  folder.node_id.identifier_type = opcua_node_encoding_numeric;
  folder.node_id.numeric_identifier = create_numeric_as_nodeid();

  /* Create and initialise folder name. If you ever delete the node from the server
   * you will need to free this data structure yourself by calling UAServer_Free.
   */
  UA_UTF8_string_t* my_folder_name = UAServer_Malloc(sizeof(UA_UTF8_string_t));
  if (!my_folder_name)
  {
    printf("UAServer_Malloc failed\n");
    return;
  }
  my_folder_name->data = (uint8_t*)"My Folder";
  my_folder_name->length = strlen((const char*)my_folder_name->data);

  folder.display_name_handle = (uint32_t)my_folder_name;

  status = UAServer_Create_Folder(&folder);
  if (status != 0)
  {
    printf("UAServer_Create_Folder returned: %d\n", (uint16_t)status);
  }
  status = UAServer_Add_to_objects_folder(folder.node_handle);
  if (status != 0)
  {
    printf("UAServer_Add_to_objects_folder returned: %d\n", (uint16_t)status);
  }

  /* Create and initialise variable. If you ever delete the node from the server
   * you will need to free this data structure yourself by calling UAServer_Free.
   */
  Integer_data_t* my_integer = UAServer_Malloc(sizeof(Integer_data_t));
  if (!my_integer)
  {
    printf("UAServer_Malloc failed\n");
    return;
  }
  my_integer->value = 0;
```

```
my_integer->source_timestamp = 0;
my_integer->status_code = OpcUa_Good;

/* Create and initialise variable name. If you ever delete the node from the server
 * you will need to free this data structure yourself by calling UAServer_Free.
 */
UA_UTF8_string_t* my_integer_name = UAServer_Malloc(sizeof(UA_UTF8_string_t));
if (!my_integer_name)
{
  printf("UAServer_Malloc failed\n");
  return;
}
my_integer_name->data = (uint8_t*)"My Integer";
my_integer_name->length = strlen((const char*)my_integer_name->data);

//Add Variable
UA_BaseDataVariable_t variable;
UAServer_Init_config_structure_BaseDataVariable(&variable);

variable.base_config.node_handle =         (uint32_t)my_integer;
variable.base_config.display_name_handle = (uint32_t)my_integer_name;
variable.base_config.node_id.identifier_type = opcua_node_encoding_numeric;
variable.base_config.node_id.numeric_identifier = create_numeric_as_nodeid();
variable.value_type =                      UA_TYPE_Int32;
variable.user_write_permissions =        0xFFFF;

status = UAServer_Create_BaseDataVariable(&variable);
if (status != 0)
{
  printf("UAServer_Create_BaseDataVariable returned: %d\n", (uint16_t)status);
  return;
}
status = UAServer_Add_to_folder(folder.node_handle, variable.base_config.node_handle);
if (status != 0)
{
  printf("UAServer_Add_to_folder returned: %d\n", (uint16_t)status);
  return;
}

/* Create and initialise variable. If you ever delete the node from the server
 * you will need to free this data structure yourself by calling UAServer_Free.
 */
String_data_t* my_string = UAServer_Malloc(sizeof(String_data_t));
if (!my_string)
{
  printf("UAServer_Malloc failed\n");
  return;
}
my_string->current_length = 0;
my_string->source_timestamp = 0;
my_string->status_code = OpcUa_Good;

/* Create and initialise variable name. If you ever delete the node from the server
 * you will need to free this data structure yourself by calling UAServer_Free.
 */
UA_UTF8_string_t* my_string_name = UAServer_Malloc(sizeof(UA_UTF8_string_t));
if (!my_string_name)
{
  printf("UAServer_Malloc failed\n");
  return;
}
my_string_name->data = (uint8_t*)"My String";
```

```
  my_string_name->length = strlen((const char*)my_string_name->data);

  //Add Variable
  UAServer_Init_config_structure_BaseDataVariable(&variable);

  variable.base_config.node_handle =         (uint32_t)my_string;
  variable.base_config.display_name_handle = (uint32_t)my_string_name;
  variable.base_config.node_id.identifier_type = opcua_node_encoding_numeric;
  variable.base_config.node_id.numeric_identifier = create_numeric_as_nodeid();
  variable.value_type =                    UA_TYPE_String;
  variable.user_write_permissions =          0xFFFF;

  status = UAServer_Create_BaseDataVariable(&variable);
  if (status != 0)
  {
    printf("UAServer_Create_BaseDataVariable returned: %d\n", (uint16_t)status);
    return;
  }
  status = UAServer_Add_to_folder(folder.node_handle, variable.base_config.node_handle);
  if (status != 0)
  {
    printf("UAServer_Add_to_folder returned: %d\n", (uint16_t)status);
    return;
  }
}
```

The node handles provided for both the folder and the variables are unique identifiers assigned by the application. These handles are opaque to the SDK but are passed back to the application in callback functions to allow the application to identify the nodes which the server is SDK is referring to during callbacks.

The display name handles need not be unique but must be resolvable to the appropriate string by the application as they are the text handles passed to the application in the UAServer_Callback_translate() API function.

Reading and Writing Variables

The server reads and writes process data by calling the read and write callbacks. All process data is the responsibility of the application. The server does not duplicate the process data except for storing sample queues and retransmission queues as per the requirements of the standard. The application is free to store its process data in any way and no assumptions are made about how this data is stored. Each data point has an associated data quality or status code. Legal values are defined in the header file opcua_status_codes.h. Each data point has a source timestamp. This timestamp should be calculated and stored at the point the data is actually read from the data source.

Reading Variables

An example read callback is shown below.

```
void UAServer_Callback_read(uint32_t handle,
                    UA_Value_type_t value_type,
                    uint32_t max_array_length,
                    uintptr_t read_context)
{
  int64_t source_timestamp = 0;
  uint32_t status_code = OpcUa_Good;
  UA_Value_t data_value;

  data_value.value_Any = 0;

  switch (value_type)
  {
  case UA_TYPE_Int32:
  {
    Integer_data_t* datum = (Integer_data_t*)handle;
    source_timestamp = datum->source_timestamp;
    status_code = datum->status_code;
    data_value.value_Int32 = &datum->value;
    UAServer_Perform_read(read_context, data_value, source_timestamp, status_code);
  }
    break;

  case UA_TYPE_String:
  {
    String_data_t* datum = (String_data_t*)handle;
    source_timestamp = datum->source_timestamp;
    status_code = datum->status_code;
    UA_Var_string_t value;
    value.data = datum->string_data;
    value.length = datum->current_length;
    value.max_length = MAX_STRING_LEN;
    data_value.value_String = &value;
    UAServer_Perform_read(read_context, data_value, source_timestamp, status_code);
  }
    break;

  default:
    status_code = OpcUa_BadInvalidArgument;
    UAServer_Perform_read(read_context, data_value, source_timestamp, status_code);
    break;
  }
}
```

The read callback provides the application with the node handle of the variable we wish to read. The value type of the variable and the expected maximum array length are provided to add type safety to the interface if the application developer requires this information. The maximum array length is specified on node creation and is zero for scalars. An opaque read_context parameter is also provided to facilitate the read.

The server is a simple single threaded implementation and as such all callbacks must completed quickly to ensure good server performance and low latency. Where the source data can be read immediately the physical read can be done in the callback function. Where the physical read would take too long, for example if it involved serial communication to a data source, then the read should be performed asynchronously and the latest value cached in the application for immediate provision to the SDK in the callback.

The actual read is performed by the SDK in the UAServer_Perform_read function call. This is done to insulate the application from the complexities of OPC UA and to allow the application to construct whatever temporary variables it requires on the stack in order to facilitate the read.

Writing Variables

Only nodes created with the appropriate user write permissions can be written to. The default node configuration is read only and the node configuration must be explicitly set to enable writing a node. An example write callback is shown below.

```
void UAServer_Callback_write(uint32_t handle,
                             UA_Value_type_t value_type,
                             UA_Status_t status_code,
                             uint32_t max_array_length,
                             uintptr_t write_context)
{
  UA_Value_t data_value;
  uint32_t new_array_length = 0;
  int64_t* source_timestamp = 0;

  data_value.value_Any = 0;

  switch (value_type)
  {
  case UA_TYPE_Int32:
  {
    Integer_data_t* datum = (Integer_data_t*)handle;
    source_timestamp = &datum->source_timestamp;
    datum->status_code = status_code;
    data_value.value_Int32 = &datum->value;
    UAServer_Perform_write(write_context, data_value, source_timestamp, FALSE, &new_array_length);
  }
    break;

  case UA_TYPE_String:
  {
    String_data_t* datum = (String_data_t*)handle;
    source_timestamp = &datum->source_timestamp;
    datum->status_code = status_code;
    UA_Var_string_t value;
    value.data = datum->string_data;
    value.length = datum->current_length;
    value.max_length = MAX_STRING_LEN;
    data_value.value_String = &value;
    if (UAServer_Perform_write(write_context, data_value, source_timestamp, FALSE, &new_array_length) == OpcUa_Good)
    {
      datum->current_length = value.length;
    }
  }
    break;

  default:
    status_code = OpcUa_BadInvalidArgument;
    UAServer_Perform_write(write_context, data_value, source_timestamp, FALSE, &new_array_length);
    break;
  }
}
```

The write callback provides the application with the node handle of the variable we wish to write. The value type of the variable and the expected maximum array length are provided to add type safety to the interface. The maximum array length is specified on node creation and is zero for scalars. An opaque write_context parameter is also provided to facilitate the write.

The server is a simple single threaded implementation and as such all callbacks must completed quickly to ensure good server performance and low latency. Where the physical write can be performed immediately the physical write can be done in the callback function. Where the physical write would take too long, for example if it involved serial communication to a data source, then the write should be performed asynchronously. In this case, set the completes_asynchronously parameter of the perform write function call to TRUE and perform the physical write asynchronously. This is the most common use case.

Where the physical write can be performed synchronously with no performance penalty then this can be done immediately after UAServer_Perform_write() when the server has written the new value and source timestamp into the temporary locations provided by the application. If the physical write is unsuccessful, the application can modify the operation level result code to indicate a failure mode.

Application Defined Node Ids

By setting a macro in the configuration header file the application developer can enable application defined node ids. Every node in the address space has a node id which is a unique identifier for a node. It is returned by a browse operation and can then be used to read, write and perform other operations on the node. A node id can be treated as a black box by a Client or it may choose to display the node id in its user interface.

Node ids internally can contain a numeric value, a GUID, a UTF8 string or a byte string (also known as opaque). When you create a node in the server address space by default the server assigns an opaque node id to the node. This byte string encodes the node handle amongst other things to optimise the speed at which the server can perform operations on the node.

The OPC UA Specification states that node ids must always be persistent. That is, they must be the same following a server power cycle for example. The opaque node id created by the server for a particular node will always be exactly the same so long as the node handle provided by the application on creation is the same.

For some applications it may be preferable to use another type of node id representation rather than opaque. In this case simply set the appropriate macro and provide the alternative representation via the node_id field of the node configuration structure base struct (UA_Base_t). Once this is done the SDK will now use the provided node id rather than the internally generated opaque one.



Warning: The SDK does not validate that user defined node ids provided by the application are unique. For correct server operation node ids must be unique. It is the responsibility of the application developer to ensure uniqueness of user defined node ids.



Warning: Application defined node ids use more memory and slow down your server. Only use this feature if it is appropriate for your platform.

File Transfer

The SDK supports file transfer according to Appendix C of Part 5 of the OPC UA Specification. The file transfer mechanism works by creating file nodes in the address space. These nodes can map to actual files in a file system or they may be logical files, i.e. data that the application wants to represent as a file but that are not stored as a file locally.

The API is clearly described in ua_server.h and a complete file transfer example is provided in the Linux examples that are packaged with the SDK. For this reason we will just touch on the highlights here rather than describe the API in detail.

The API is comprised of a number of callback functions that the application must implement. These callbacks perform a number of well-known file operations as follows:

- Get the size of a file
- Open a file
- Read from a file
- Write to a file
- Get the current file position
- Set the current file position
- Close a file

The functionality is implemented in a defensive manner to allow the application developer to map these API calls directly to native file IO operations. In built protections are:

- All file handles provided by a Client are validated for correctness before being passed to the API callbacks. Specifically we check that the file handle is known (was returned by a previous Open call) and that the file is currently open.
- If a session closes without the Client closing all open files the SDK will call back the application to close any open files.

If the application closes files independently of the SDK API callbacks then care must be taken in the API callbacks to detect an access request to a previously closed file and handle this gracefully by returning an appropriate error message.



Warning: The OPC UA file transfer specification is similar to UNIX file IO in that there is no distinction between text and binary files and all files are treated as binary files. When using the API on a Windows platform all files should be opened in binary mode.

Deferred Reads

For ease of integration the SDK is implemented as synchronous and single threaded. While this makes it very easy to use it also places restrictions on the application. The application must respond immediately to any API callbacks or else the entire server blocks, monitored items cannot be sampled and all Clients will have to wait for response messages.

Where the server data source is local to the server CPU or is available over a very fast serial link (such as 40 Mbps SPI) this model works perfectly and the application can synchronously read from and write to the data source. Where the data source is at the end of a latent or slow link this requires a different approach.

In the case of a latent or slow data source the application can optimise its operation by using two mechanisms. These mechanisms allow the application to distinguish between periodically sampled data and asynchronously read data.

Whenever a Client creates, modifies or deletes a monitored item in the server the SDK calls the application back with UAServer_Callback_monitored_item(). This provides the application with all of the information it requires in order to configure periodic sampling of the data source at the required rate.

Whenever a UAServer_Callback_read() occurs the application can identify whether the read is a result of a sampling monitored item or of an asynchronous read request from a Client by calling UAServer_Is_read_operation_asynchronous().

Where it is a periodic sample the application simply provides the most recent value it has sampled from the data source.

Where it is an asynchronous read the application can defer the readoperation by calling UAServer_Defer_read_operation(). When the value is available the application signals the SDK by calling UAServer_Read_deferral_complete().

If the session closes or the read operation is no longer valid for some other reason the SDK will callback the application with UAServer_Callback_deferred_read_operation_aborted().



Warning. This is an advanced feature and the deferral handle passed between the SDK and the application is a pointer to a deferral object. If the SDK has called UAServer_Callback_deferred_read_operation_aborted() for a deferral_handle the application must not call UAServer_Read_deferral_complete() for that deferral_handle.



Warning. The application must not call UAServer_Read_deferral_complete() from within a callback function.

Asynchronous Operation

For ease of integration the SDK is implemented as synchronous and single threaded. While this makes it very easy to use it also places restrictions on the application when security is enabled. OPC UA security makes use of private key RSA operations that are quite slow. On a 168Mhz ARM Cortex-M4F a private key RSA operation can take 400ms with a key size of 1024 bits. If this is performed synchronously it can increase server latency and cause missed samples when sampling monitored items at high rates.

For these reasons the SDK offers the option of offloading these operations to a worker thread so as not to delay the main thread of operation. The API that enables this functionality is simple consisting of seven functions.

When the feature is enabled and the SDK wishes to offload an operation it calls UAServer_Callback_process_asynchronously(). The application should pass the handle provided to a worker thread. In the worker thread UAServer_Process_asynchronously() is called with the handle as the argument. When the function has returned the worker thread should pass the handle back to the main thread. The main thread should then call UAServer_Operation_completed() with the handle as the argument.

The application must also declare a mutex type for the platform and implement the callback functions UAServer_Callback_mutex_init(),UAServer_Callback_mutex_destroy(), UAServer_Callback_mutex_lock() and UAServer_Callback_mutex_unlock().



Warning. This is an advanced feature and the operation handle passed between the SDK and the application is a pointer to an object. The application must always complete the operation or the behaviour is undefined.



Warning. When shutting down the server the application must ensure that all asynchronous operations have completed before terminating the server.

Integrating Security

The server 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 in available via MatrixSSL integration. In order to integrate security set the relevant macros in the configuration header to 1 and link to your chosen cryptography library. Linux and Windows examples are provided as part of the SDK distribution.

OPC UA Security Concept

OPC UA provides security by authenticating clients and servers and encrypting communications using X.509v3 application instance certificates. Public key algorithms are used for symmetric key generation and exchange periodically, with the majority of 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 was generated elsewhere.

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

Certificate Stores

Servers typically maintain certificate stores in the file system. Each server has four stores. For secure channel management a trust store and issuer store are maintained. Similarly for optional user authentication a trust store and issuer store are maintained.

When a client attempts to connect to a server the certificate stores are searched and a chain of trust is validated. The chain must include all certificates from the client certificate up to and including the root CA certificate. At least one certificate in the chain must reside in the trust store rather than issuer store.

Upon negotiating a secure channel, if a client certificate is valid but not trusted, or if the negotiation fails for any reason that can be suppressed via the API, the client certificate is placed in the application trust store in the certs/rejected directory. Moving the certificate up one level into the certs directory adds the certificate to the trust list.

The MatrixSSL integration module provided in the API is implemented assuming the availability of a file system. File systems are easily implemented on microcontrollers but if you would rather avoid a file system entirely and wish to use another storage mechanism the integration module will have to be refactored to replace the file access functionality.

Endpoint Configuration

A server endpoint defines a logical point to which a client can attempt to connect. A single server can have multiple endpoints implementing different security policies. Each endpoint may also accept a number of different user token types (anonymous, username/password, etc.). Each endpoint type can be enabled in the configuration by adding one or more user token policies to the endpoint configuration. This is done via a bitmap of user token policies. Setting the user token policy bitmap to zero for a particular endpoint completely disables it.

Initial Provisioning



Warning. Every server must possess a unique application instance certificate. The device vendor must decide how to fulfil this requirement. There are three options.

1. Have the SDK create 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 advantages of this approach are that it requires little intervention by the end user and the certificate private key is not exposed to a 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 lived and the strength of the security they provide is only as strong as the strength of the key pair used to sign them. Also the certificate generation can take a long time and this will influence initial device boot time.

2. Create a default, self-signed certificate on the production line and install it in the device.

   The advantages of this approach are 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.

3. Ship the device with no certificate and oblige the end user 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 in order to use CA issued certificates which may suit his use case better.

Trusting All Certificates

In order to facilitate initial provisioning the application can call UASecurity_Set_validation_options() with the option bit OB_TRUST_ALL_CLIENTS_WHEN_TRUST_LIST_IS_EMPTY (bit 31) set. Doing so forces the server to trust all Client certificates when there are no Client certificates in the trust list. This facilitates connecting initially with a configuration Client in order to configure and provision the server.

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 layer where the configuration Client has to provide user credentials such as a unique or one-time administrator password in order to complete the connection.

Self-signed versus CA Issued Certificates

Self-signed

- No need for CRLs (Certificate Revocation Lists)
- Device trust list must be updated for every new application that requires a connection
- Device trust list can become very large over time increasing Flash and RAM requirements

CA issued

- Only a single CA certificate 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 now become important and the device CRL(s) should be updated regularly

Security Error Suppression

Certain certificate errors may be suppressed by the application using the API. Specifically the application can suppress certificate validity errors (time) and whether or not he use of Certificate Revocation Lists is required. 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:

- How to handle expired certificates (both our application certificate and those of applications we are connected to)
- Whether or not to enforce hostname validation
- Whether or not to require use of CRLs



Warning. Incorrectly set policies can have negative consequences such as a 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 to 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 up one level in the file system to the certs folder.

On very small systems this has the potential to exhaust the file system storage over time and so this functionality can be disabled by setting the appropriate validation option bit.

Security API

Server security configuration is very simple. The security API is less than 10 functions, all of which are optionally called by the application.

UASecurity_Check_application_certificate_exists()

Check if the server application instance certificate and private key file are present.

UASecurity_Create_self_signed_certificate()

Create a default self-signed application instance certificate and corresponding RSA key pair.

UASecurity_Reload_directory_stores_now()

The application has changed the contents of the directory stores and instructs the SDK to reload the stores from the file system.

UASecurity_Reload_directory_stores_before_each_validation()

The directory stores can be modified at any time via an out of band mechanism such as SFTP, another process, etc. For this reason reload the stores from the file system on every Client certificate validation.

UASecurity_Create_directory_stores()

Create the directory certificate stores folder structure in the file system.

UASecurity_Set_validation_options()

Set the certificate validation options. The available options are as follows:

```
/** Option Bit - Accept expired certificates */
#define OB_SUPPRESS_CERTIFICATE_EXPIRED      (1<<0)
/** Option Bit - Do not validate the application hostname */
#define OB_SUPPRESS_HOSTNAME_INVALID         (1<<1)
/** Option Bit - If checking certificate CRLs do not reject
 * a certificate if a CRL for the Issuer cannot be found */
#define OB_SUPPRESS_REVOCATION_STATUS_UNKNOWN (1<<2)
/** Option Bit - Look for CRLs online (not currently supported) */
#define OB_CHECK_REVOCATION_STATUS_ONLINE     (1<<3)
/** Option Bit - Look for CRLs locally (typically in the local filesystem */
#define OB_CHECK_REVOCATION_STATUS_OFFLINE    (1<<4)
/** Option Bit - Use default options for the PKI provider */
#define OB_USE_DEFAULT_OPTIONS               (1<<5)
/** Option Bit - Trust all client certificates if the trust list is empty */
#define OB_TRUST_ALL_CLIENTS_WHEN_TRUST_LIST_IS_EMPTY   (1U<<31)
```

UASecurity_Get_thumbprint()

A utility function to calculate the thumbprint of a certificate in case the application wished to add the certificate to a directory store. Certificates in the directory store should be stored with the thumbprint as the filename to ensure uniqueness. Certificates must be stored in DER format.

UASecurity_Callback_get_private_key_password()

The SDK will store the application instance certificate private key filename in an encrypted format. When accessing this file the SDK requests the password from the application.

UASecurity_Callback_certificate_added_to_rejected_list()

A Client attempted to connect to the server. The Client application instance certificate was valid but untrusted. For that reason the SDK has added the certificate to the rejected folder of the application's trusted directory store.

OpenSSL Integration

OpenSSL integration is straightforward. On a Linux platform all that has to be done is for OpenSSL to be installed if it is not present already and for the application to be linked to the OpenSSL library.

On a Windows platform the Visual C++ 2008 Example will only build if OpenSSL is installed at C:\OpenSSL-Win32. Installers can be obtained from at your own risk from **https://slproweb.com/products/Win32OpenSSL.html**.

Entropy Generation

The OpenSSL interface module provided with the SDK attempts to obtain entropy during initialisation 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 your device with a certain hardware configuration but when you make changes it may stop working. Changes could be the removal of a keyboard or network interface for example.

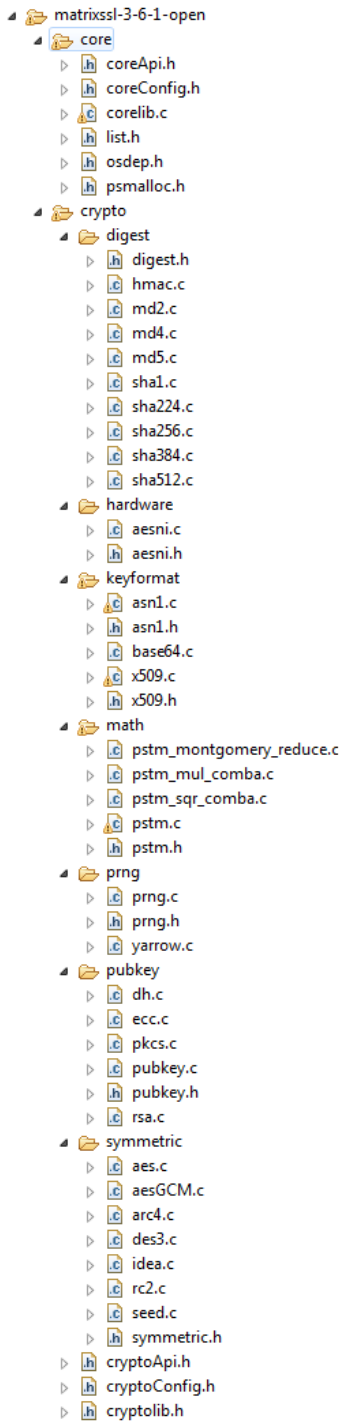
It is the responsibility of the device vendor and application developer to decide what is the most appropriate entropy source on their system.



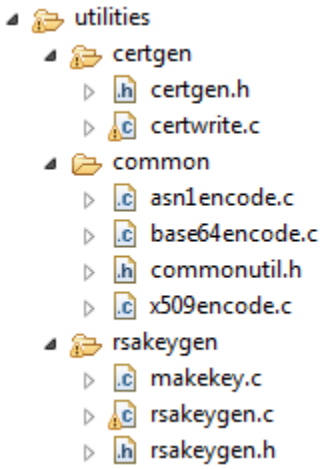
Warning. Operating a server with poor entropy will directly affect the level of security obtained. Both RSA (certificate) and secure channel symmetric key generation require appropriate levels of entropy to provide the expected level of security.

MatrixSSL Integration

Please note that the HDK (Hardware Development Kit) security example uses MatrixSSL but MatrixSSL is not provided as part of the project. In order to build the project successfully add the following MatrixSSL files (downloadable from their website) and modify them as per the instructions below. If you would like to save time by obtaining the pre-modified files please contact MatrikonOPC support. The files required for basic integration are shown below.



The files required for implementing certificate generation (which is optional but recommended) are not contained within the GPL version downloadable from their website. These files are available directly from MatrixSSL and are shown below.



The files required for implementing Basic256 are not contained within the GPL version downloadable from their website. These files are available directly from MatrixSSL.

MatrixSSL integration is quite involved and requires many steps. As with most commercial SSL libraries for embedded systems the public API they expose is for SSL only. OPC UA Secure Conversation requires us to integrate with these libraries via internal APIs that were not designed to be publicly accessible and for this reason a number of modifications are required. These steps are documented in the API module ua_security_matrixsssl.c and are reproduced below.

Copyright 2014 MatrikonOPC. All rights reserved.

This module was implemented for integration with MatrixSSL version 3.6.1 and tested with this version.
Usage with other versions may require modifications to the source code below.

In order to use this module with MatrixSSL the following tasks must be undertaken:

Implement the following macros in MatrixSSL file coreConfig.h

```
#define HAVE_NATIVE_INT64
#define MATRIX_USE_FILE_SYSTEM
```

Uncomment the following macro in MatrixSSL file cryptoConfig.h

```
#define USE_CRL
```

If using a POSIX OS Implement the following macros in MatrixSSL file psmalloc.h. Otherwise implement the memory allocation mapping in matrixos.h.

```
#define psMalloc(A, B)   UAServer_Malloc(B)
#define psRealloc        UAServer_Realloc
#define psFree           UAServer_Free
```

Implement the platform abstraction layer osdep.c as outlined the MatrixSSL documentation.
If required #define PS_UNSUPPORTED_OS in MatrixSSL file osdep.h and implement the abstraction layer header file matrixos.h

Example platform layers are provided by MatrikonOPC.

Implement a file system on your platform. A file system is not required but this MatrixSSL integration module assumes the presence of a file system. If you do not wish to use a file system changes will be required to this module.

Always include <time.h> and remove POSIX macro around function validateDateRange() in MatrixSSL file x509.c.

Remove the following code from psX509ParseCert() in MatrixSSL file x509.c as date validation is done during certificate chain authentication and is not required during certificate parse / import.

```
if ((rc = validateDateRange(cert)) < 0) {
  psTraceCrypto("Validity date check failed\n");
  return rc;
}
```

Change the code in parseGeneralNames() in MatrixSSL file x509.c from

```
if (*c <= ' ' || *c > '~') {
```

to

```
if (*c < ' ' || *c > '~') {
```

Change the function signature of validateDateRange() in MatrixSSL file x509.c from
static int32 validateDateRange(psX509Cert_t)
to
int32 validateDateRange(psX509Cert_t)
and move the function prototype from x509.c to x509.h so it is accessible from this module.

In function psX509AuthenticateCert() in MatrixSSL file x509.c add a test for not self signed around the CA permissions BasicConstraints test (as shown below) as we do not require a self signed certificate to have CA permissions. In fact it is undesirable that a self signed application instance certificate would have such permissions.

```
if ((ic->version > 1) && (ic->extensions.bc.cA <= 0)) {
  psTraceCrypto("Issuer does not have basicConstraint CA permissions\n");
  if (sc != ic){
    sc->authStatus = PS_CERT_AUTH_FAIL_BC;
    return PS_CERT_AUTH_FAIL_BC;
  }
}
```

In the function pkcs1Unpad() in MatrixSSL file pkcs.c do not require the caller to specify the plaintext length. From RFC2313, "For block types 01 and 02, the encryption block can be parsed unambiguously since the padding string PS contains no octets with value 00 and the padding string is separated from the data D by an octet with value 00".

Therefore modifying the the plaintext length test as follows:

```
//if ((uint32)(end - c) != outlen) {
//  psTraceCrypto("pkcs1Unpad verification failure\n");
//    return PS_LIMIT_FAIL;
//  }

outlen = (uint32)(end - c);
```

If including certificate generation, add the following code to support additional key usage types in function psX509ParseCertConfigBin in MatrixSSL file x509encode.c

```
if (strstr((char*)&fileBin[valIndex], "digitalSignature") != NULL) {
  certConfig->ext->keyUsageFlags |= KEY_USAGE_DIGITAL_SIGNATURE;
}
if (strstr((char*)&fileBin[valIndex], "keyEncipherment") != NULL) {
  certConfig->ext->keyUsageFlags |= KEY_USAGE_KEY_ENCIPHERMENT;
}
if (strstr((char*)&fileBin[valIndex], "dataEncipherment") != NULL) {
  certConfig->ext->keyUsageFlags |= KEY_USAGE_DATA_ENCIPHERMENT;
}
if (strstr((char*)&fileBin[valIndex], "nonRepudiation") != NULL) {
  certConfig->ext->keyUsageFlags |= KEY_USAGE_NON_REPUDIATION;
}
```

If including certificate generation, in function psWriteCertExt() in MatrixSSL file x509encode.c always write the basic constraints by modifying the code as follows:

```
//if (1) { ext->bc.cA != 0) {
  rc = psWriteCertBasicConstraints(pool, &bcBuf, ext->bc.cA,
  ext->bc.pathLenConstraint, &bcLen);
  rc = psWriteSubjectKeyId(pool, &skBuf, ext, &skLen);
//}
```

If including certificate generation, add new function psX509WriteSelfSignedCertDer() to MatrixSSL file certwrite.c (function can be obtained from MatrikonOPC).

If including certificate generation, in function psRsaWritePrivKeyFile() in MatrixSSL file rsakeygen.c and in function psBase64EncodeAndWrite() in MatrixSSL file base64encode.c change the following:

FILE to OS_FILE
fopen to os_fopen
fwrite to os_fwrite
fclose to os_fclose

Runtime Configuration Settings

SDK build configuration is controlled via build macros defined in ua_server_config.h. In addition to the build configuration the SDK must be configured at runtime by passing in a configuration data structure to UAServer_Initialise(). The configuration data structure is very important as it places limits on what the server will allow a Client to do. These limits are essential to preventing out of memory conditions at runtime. Other limits prevent operations that may result in very high CPU utilisation.

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

The configuration data structure is defined in ua_types.h and an example structure is provided in ua_server_config_template.c. The configuration fields are outlined below.

TCP Binary Stack Layer

HELLO_TIMEOUT_IN_MS

The allowed time between a Client opening a TCP socket with the server and the server receiving a hello message. Default value is 60 s and this does not need to be changed.

CONNECTION_TIMEOUT_IN_MS

The allowed time between messages from the Client. Default value is 60 s but the actual required value is application specific and depends on the expected message frequency for your application. The server will close 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.

PORT_NUMBER

The TCP port that the application is listening on for incoming Client connections. Port 4840 is the standard port.

MAX_MESSAGE_SIZE

The maximum message size in bytes that the server will accept from a Client. This must be at least as big as a single message chunk. Typical values are:

- 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 (we recommend 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.

MAX_MESSAGE_CHUNK_SIZE

The maximum message chunk size is most significant when using security as the overall message is chunked into pieces which 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 small server.

MAX_MESSAGE_CHUNK_COUNT

The maximum number of message chunks allowed to make up a message. This number multiplied by the message chunk size must be >= the maximum message size.



Warning. Ensure via testing that the messages your 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.

MAX_CONNECTIONS

The maximum number of TCP Binary connections that the server supports. This value must equal at least 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 your application requires.

Secure Conversation Stack Layer

MIN_REVISED_CHANNEL_LIFETIME_IN_MS
MAX_REVISED_CHANNEL_LIFETIME_IN_MS

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 will accept. If the Client requests a lifetime outside of these limits then the server will restrict the negotiated lifetime to the appropriate limit.



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 $*10^{18}$ AES keys per second would require about 3×10^{51} 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.

MAX_SECURE_CHANNELS

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 your application requires.

Security

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:

- 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

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.

To disable an endpoint security policy completely enter zero for the field value. If you want 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 X509v3 Certificate - Token Security Policy None
- User Token X509v3 Certificate - Token Security Policy Basic128Rsa15
- User Token X509v3 Certificate - Token Security Policy Basic256

If using user token username / password for unencrypted endpoint security policies it is recommended to only enable user token policies that 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 it is recommended to enable username / password - token security policy none.

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

Services

MIN_REVISED_SESSION_TIMEOUT_IN_MS
MAX_REVISED_SESSION_TIMEOUT_IN_MS

The session service negotiates a revised session timeout with a Client when the Client connects to the server. These settings place bounds on the allowable range that the server will accept. If the Client requests a timeout outside of these limits then the server will restrict the negotiated timeout to the appropriate limit.



Warning. In general it is 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 Clients close sessions gracefully on disconnect but in the case of a Client terminating a connection without closing the session we want the session to timeout in a reasonable period of time.

Also, the SDK will recycle secure channels that have no active session associated with them. This gives us a means of reusing both sessions and secure channels that are inadvertently left open by a misbehaving Client.

If your application allows it keep session timeouts much smaller than secure channel timeouts to enable session and secure channel recycling.

The default values are both 60 s but the actual required values are application specific.

MAX_SESSIONS

The maximum number of sessions that the server supports. This value must equal at least the number of concurrent client connections required.



Warning. Sessions require significant amounts of memory. For this reason it is important not to make this value any larger than your application requires.

MAX_SUBSCRIPTIONS_PER_SESSION

The maximum number of subscriptions supported per session. Subscriptions are the server component that report 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, only the rate at which they are reported.

Where the server supports sample queues the server can support multiple monitored items sampling at different rates within a single subscription sampling at a slower rate.

Where 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 your application requires.

MIN_MAX_KEEP_ALIVE_COUNT
MAX_MAX_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 will accept. If the Client requests a count outside of these limits then the server will restrict 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 publish requests from the Client. The values for these limits are application specific. Unless you have a specific reason to restrict the Client in its choice of max keep alive count it is recommended to keep these limits very wide.

MAX_NOTIFICATION_RETRANSMISSION_QUEUE_SIZE

Clients may create subscriptions to sample data where they only require the latest live value. In other cases a Client may wish to create a subscription where they want to log all sampled values 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 in order 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 your application requires.

MAX_PUBLISH_REQUESTS_PER_SESSION

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 dequeues the request and sends a publish response containing data change notifications to the Client.

In order to allow timely reporting over high latency networks the server has the capacity to queue multiple publish requests internally. 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. Queueing publish requests requires significant amounts of memory. For this reason it is important not to make this value any larger than your application requires.

BROWSE_MAX_REFS_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 but also slows down the brose 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.

MAX_MONITORED_ITEMS

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 your application requires.

MAX_MON_ITEM_QUEUE_SIZE

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 then 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 your application requires.

Informational Arrays

NAMESPACE_ARRAY

Nodes created in the server address space can be defined as belonging to a namespace. There are two default namespaces created by the SDK, these are the OPC UA standard namespace (index 0) and the server namespace (index 1). If any additional namespaces are required they can be added here beginning at index 2. The appropriate namespace can be specified for a node by defining the index when creating the node.

SOFTWARE_CERTIFICATES

Software certificates are a feature of the OPC UA standard that are specified but not currently implemented by the OPC Foundation and vendors. In future, should they be implemented they can be exposed in the server here.

SERVER_PROFILE_ARRAY

If a server is designed to support certain combinations of features called profiles, the supported profiles can be exposed in the server here.

LOCALE_ID_ARRAY

The server can support translation of text into multiple languages. If the application requires to support this it must add an array of supported locales in order of preference and resolve the locale array index and text handle to UTF8 text strings in the relevant translate callback. The default locale array contains English only without a country code "en".

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.

MAX_BROWSE_CONTINUATION_POINTS

The maximum number of continuation points supported by the browse service. Default value is 100.



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 your application requires.

MAX_QUERY_CONTINUATION_POINTS
MAX_HISTORY_CONTINUATION_POINTS

Query and History are not currently supported by the SDK.

MAX_ARRAY_LENGTH

The maximum array length supported by the server. Default value is 128 bytes but the actual value is application specific.

MAX_STRING_LENGTH

The maximum UTF8 string length supported by the server. 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.

ENFORCE_OPERATION_LIMITS

Set to TRUE to enforce limits. Enforcing limits protects against out of memory conditions but may affect interoperability with Clients that do not observer these values. Default value is FALSE.



Warning. If enforcing limits make sure you carefully test interoperability against all Clients you intend to interoperate with.

MAX_NODES_PER_READ

Maximum nodes to read in a read service call. Default value is 100.

MAX_NODES_PER_HISTORY_READ_DATA
MAX_NODES_PER_HISTORY_READ_EVENTS
MAX_NODES_PER_HISTORY_UPDATE_DATA
MAX_NODES_PER_HISTORY_UPDATE_EVENTS

History is not currently supported by the SDK. Default is 0.

MAX_NODES_PER_WRITE

Maximum nodes to write in a write service call. Default value is 100.

MAX_NODES_PER_METHOD_CALL

Maximum nodes per call service call. Default value is 100.

MAX_NODES_PER_BROWSE

Maximum nodes per browse service call. Default value is 100.

MAX_NODES_PER_REGISTER_NODES

Maximum nodes per register nodes service call. Default value is 100.

MAX_NODES_PER_TRANSLATE_BROWSE_PATHS_TO_NODE_IDS

Maximum nodes per translate browse paths to node ids service call. Default value is 10.

MAX_NODES_PER_NODE_MANAGEMENT

Node management is not currently supported by the SDK. Default is 0.

MAX_MONITORED_ITEMS_PER_CALL

Maximum nodes per monitored item service set service call. Default value is 100.

Internals

USE_HEAP_RATHER_THAN_RAM_BUFFER

Use the system heap for all memory allocations. The heap is called via the API callback functions UAServer_Callback_malloc() and UAServer_Callback_free(). Default is FALSE which uses the internal allocator.



Warning. Using the heap eliminates the need to size the memory buffer correctly via pre-release testing but obviously requires that the system memory is large enough to safely operate over an extended period of time. Using the internal allocator requires the application developer to stress test the application up to the configured limits to correctly size the RAM buffer.

BUFFER
BUFFER_LENGTH_IN_BYTES

When using the internal allocator for all memory allocations, provide a RAM buffer to the SDK from which it will source all required memory.

MAX_ALLOCATION_SIZE_IN_BYTES

If using the internal allocator specify the maximum allocation size in bytes. This must be as large as the maximum message size. If your application makes use of the SDK API functions UAServer_Malloc(), etc. this value must take into consideration the maximum size allocated by the application using these functions if larger than the maximum message size.

ADDRESS_SPACE_HASH_TABLE_NO_OF_BUCKETS

The address space nodes are stored in a hash table for fast look up. The number of buckets in this hash table can be specified here. A larger table reduces lookup time but increases memory utilisation. Default value is 997. For a very small server this could be as small as 11.



Warning. Hash table buckets require significant amounts of memory. For this reason it is important not to make this value any larger than your application requires.



Warning. The number of buckets in the hash tables should be set equal to a prime number. This minimises the number of bucket collisions improving server performance where table lookups are required.

MAX_NO_OF_DEFERRED_RESPONSE_MESSAGES

If using deferred read response messages this value limits the number of concurrent deferred response messages.

MAX_NO_OF_DEFERRED_OPERATIONS_PER_RESPONSE_MESSAGE

If using deferred read response messages this value limits the number of concurrent deferred read operations per deferred response messages.



Warning. Deferred read operations and response messages require significant amounts of memory. For this reason it is important not to make this value any larger than your application requires. Do not use this feature unless required by your application.

Diagnostics

Server diagnostics are used in conjunction with a Wireshark trace to obtain SDK support for any unexpected or undesirable server behaviour.

ENABLE_INFO_LOG

Enable informational log messages via the API function UAServer_Callback_log().

ENABLE_ERROR_LOG

Enable error log messages via the API function UAServer_Callback_log(). Error messages can be indicative of an internal SDK error or an invalid operation by a Client.

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. In order to minimise excessive CPU utilisation it is possible to restrict the use of percent dead band filters completely, to single precision values and to scalar variables.

DISABLE_PCT_DEADBAND

Disable monitored item percent dead band filtering for all variables.

DISABLE_PCT_DEADBAND_VECTORS

Disable monitored item percent dead band filtering for all vector variables.

DISABLE_PCT_DEADBAND_FLOATING_POINT

Disable monitored item percent dead band filtering for all floating point variables.

DISABLE_PCT_DEADBAND_DOUBLE

Disable monitored item percent dead band filtering for all double precision floating point variables.

DISABLE_PCT_DEADBAND_FLOATING_POINT_VECTORS

Disable monitored item percent dead band filtering for all floating point vector variables.

DISABLE_PCT_DEADBAND_DOUBLE_VECTORS

Disable monitored item percent dead band filtering for all double precision floating point vector variables.

Intervals

The server executes at a cyclic rate defined by the application. The SDK must be polled at this cyclic rate. There is no filtering employed internally to determine whether or not the application is polling it at the correct rate. All sampling, etc. that the server performs can only be a multiple of the server cyclic rate.

INTERVALS_ARRAY
INTERVALS_ARRAY_LENGTH

The interval array contains an array of predefined cyclic rates that the server can implement very efficiently for sampling and reporting data changes. All intervals must be a multiple of the server cyclic rates.

ALLOW_ARBITRARY_INTERVALS

Where the application does not know what cyclic rates Clients will need to use to sample data and subscribe to it, arbitrary interval scan be supported.



Warning. Arbitrary intervals are not as efficient as intervals predefined in the intervals array and can result in higher CPU utilisation. Only allow arbitrary intervals if you cannot specify the rate requirements in the intervals array.

MIN_INTERVAL_IN_MS

Minimum interval in ms. Must be <= index 0 of the intervals array.

MAX_INTERVAL_IN_MS

Maximum interval in ms. Must be >= the last entry of the intervals array.

CYCLIC_RATE_IN_MS

The cyclic rate at which the server executes and the rate at which the server is polled by the application. Default value is 100 ms. It is not recommended to increase the rate beyond this (50 ms for example) without very careful testing.



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

Memory Usage and Performance

Performance Issues

As discussed previously all callbacks must be completed quickly in order to provide good server performance and low latency operation. As also mentioned previously, 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 utilisation is important all steps must be taken to ensure efficient callback implementations.

Due to the single threaded nature of the design jitter can be introduced into the sample rates where clients are making service calls just before a scheduled sampling event. For example, where a node is being sampled by a monitored item at a 100ms period, the read callback would be expected to execute every 100ms for that node. Due to the server's need to process service calls from one or more clients the actual instant at which the read callback executes may move about slightly due to the loads on the server. In most applications a small amount of sampling jitter is unimportant but where this is unacceptable, the data sources should be sampled asynchronously and periodically (in another thread or in an interrupt) and any sampling jitter can be avoided. The source timestamp which is updated at the time of sampling will faithfully represent the sample instant and the exact time when the server reads the cached sample via the read callback becomes unimportant.

The UAServer_Callback_monitored_item() function enables the application to configure its sampling functionality to match the sample rate in use by the server for any particular variable. It is possible for a session to create multiple monitored items for the same node at various sample rates. It is also possible for multiple clients or sessions to monitor the same node at different rates. The SDK has no knowledge of the application specific requirements and for that reason the SDK will create a monitored item on request and makes no attempt to merge multiple monitored items for the same variable into a single read callback. If the application has specific requirements regarding the sampling of particular variables it can implement appropriate logic in response to the monitored item callback.

For example, let's imagine that a server is sampling a variable asynchronously over a slow serial line at a 500ms rate in response to the creation of a monitored item for that variable. When the read callback executes on average every 500ms the latest cached value is provided to the SDK. Now a new monitored item is created for the same variable but with a sample rate of 100ms. It is legal according to Part 4 of the OPC UA Specification to sample the variable at a 100ms rate and simply report the latest value in each callback thus providing the required behaviour with a single sample rate on the serial line.

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

Performance can further be improved by ensuring that SDK hash tables use a bucket count which is a prime number. This minimises the collisions when searching the tables, thereby improving performance.

For reference, refer to the following table when testing your application performance as a guide to typical server performance on a number of test architectures.

Table 2: Typical Server Performance			
Test	**Conditions**	**Hardware**	**CPU Utilisation (%)**
100 continuously changing tags	Sampling and reporting every 100ms	ARM Cortex-M4F (STM32F407) @ 168MHz	15%

*Metrics obtained using GCC -O3

Memory Utilisation

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 shall be performed from the buffer provided as part of the configuration that is passed to the UAServer_Initialise() function. In order to determine how much memory is required perform the following analysis:

1. Set the memory buffer to a very large value or as large a value as is practical on your platform.
2. Run the server for an extended period exercising it in such a way so that the maximum memory utilisation is performed (taking into account the runtime configuration limits you have set).
3. Call the memory info API function in order to obtain the recommended memory for your test run.

To ensure the SDK uses the maximum memory possible you should take note of the following factors:

1. The number of concurrently connected clients is important.
2. 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.
3. 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 behaviour in production. The bigger the data payload, the more memory each monitored item will require.
4. Where the server configuration allows for monitored item sample queue lengths greater then 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 in order to maximise 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 behaviour such as Clients being forcibly disconnected from the server, monitored item sampling failure amongst other behaviours.



Warning. 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.

Other Features and Further Information

User Groups

The default SDK configuration allows anonymous client access to the server. This can be disabled if required. You can also enable username & password based access to the server. In this case, a simple login callback is provided whereby the application validates a username and password combination provided by the SDK and returns a user index parameter to allow or disallow access and inform the stack which user group is logged in. It is up to the application to maintain username and password pairs.

When adding nodes to the address space, the node visibility can be filtered based on the logged in user. This is done via a uint16_t based bitmask. Bit zero enables node visibility for the anonymous user and bits 1 to 15 are 15 additional user groups that can be supported if required. The default value for the bitmask after initialising a configuration structure is 0xFFFF (node visible to all user groups). If you don't want to filter node visibility based on user groups or if you don't want to support user groups you can ignore this functionality completely and use the default value.

Views

Distinct views can be created. These views are a subset of the nodes and references in the address space. Views allow the server to filter the nodes displayed and their relationships to other nodes based on a view that a client specifies. Views can be supported in conjunction with or completely independent of users. A view is created by creating a view node and adding at least one other node to the view node. Clients browsing this view will see only references that are defined as visible for that view.

When adding nodes to folders (i.e. creating references between nodes), the reference visibility can be filtered based on the current view. This is done via a uint16_t based bitmask. Bit zero enables reference visibility for no views and bits 1 to 15 are 15 specific views that can be supported if required. The default value for the bitmask after initialising a configuration structure is 0xFFFF (node visible in all views). If you don't want to filter reference visibility based on views, or if you don't want to support views, you can ignore this functionality completely.

API Reference Manual

The API reference manual is available in HTML format in the SDK distribution package at "Distribution/SDK Documentation/API Reference/index.html".

Compliance Certification

If you require OPC Foundation compliance certification for your product / device you will need to undergo compliance testing in the OPC Foundation test lab. Compliance testing is free to OPC Foundation Corporate Members. Obtaining compliance certification is typically very straightforward when using the MatrikonOPC OPC UA Embedded Server SDK. The OPC Foundation have decided that the use of an SDK alone does not endow compliance certification on a product so you must still undergo certification testing.

An example application is provided that can be used to demonstrate passing of the CTT by the SDK. You can also use this application as a basis for implementing a test configuration on your own device in order to test against the CTT. Compliance testing involves more than the CTT but passing

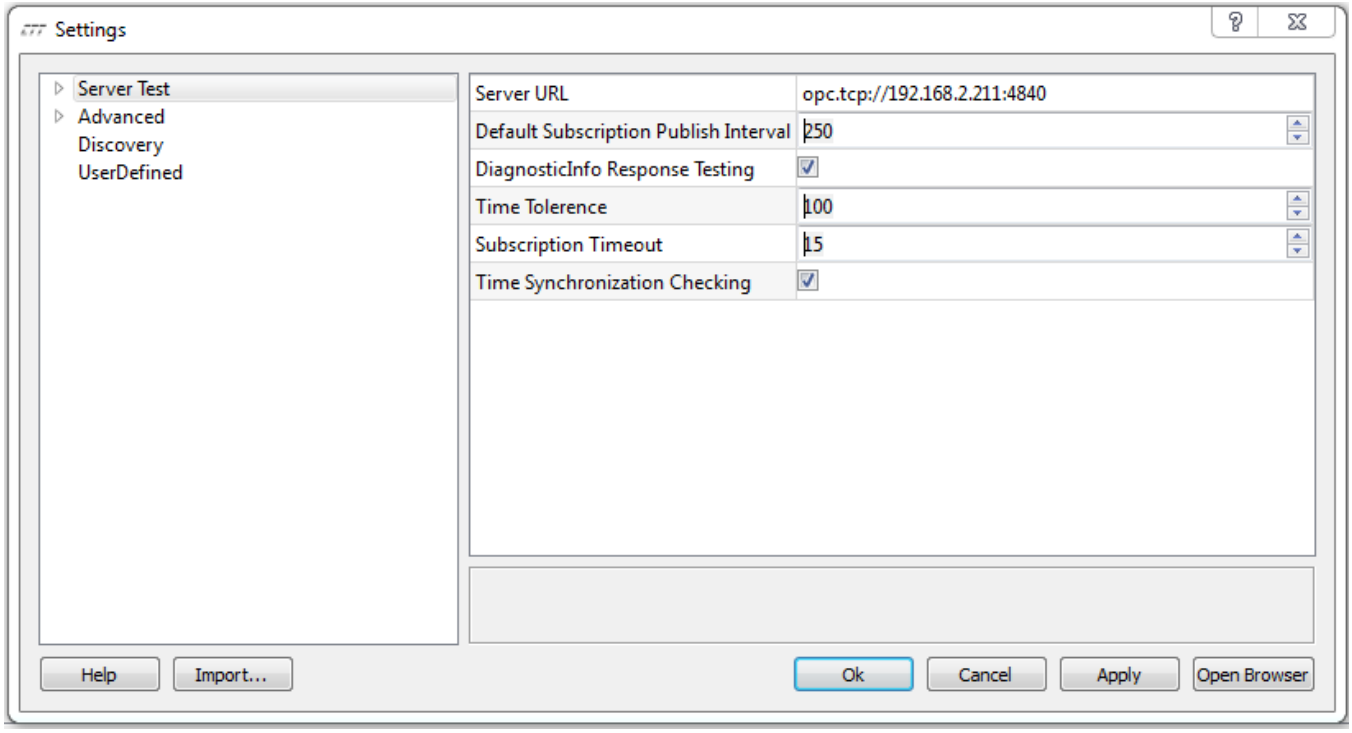
the CTT is a prerequisite for compliance testing. Version 2.0.4 passes the Embedded Server Profile and Data Access Profile test cases with CTT version 1.2.335.238.

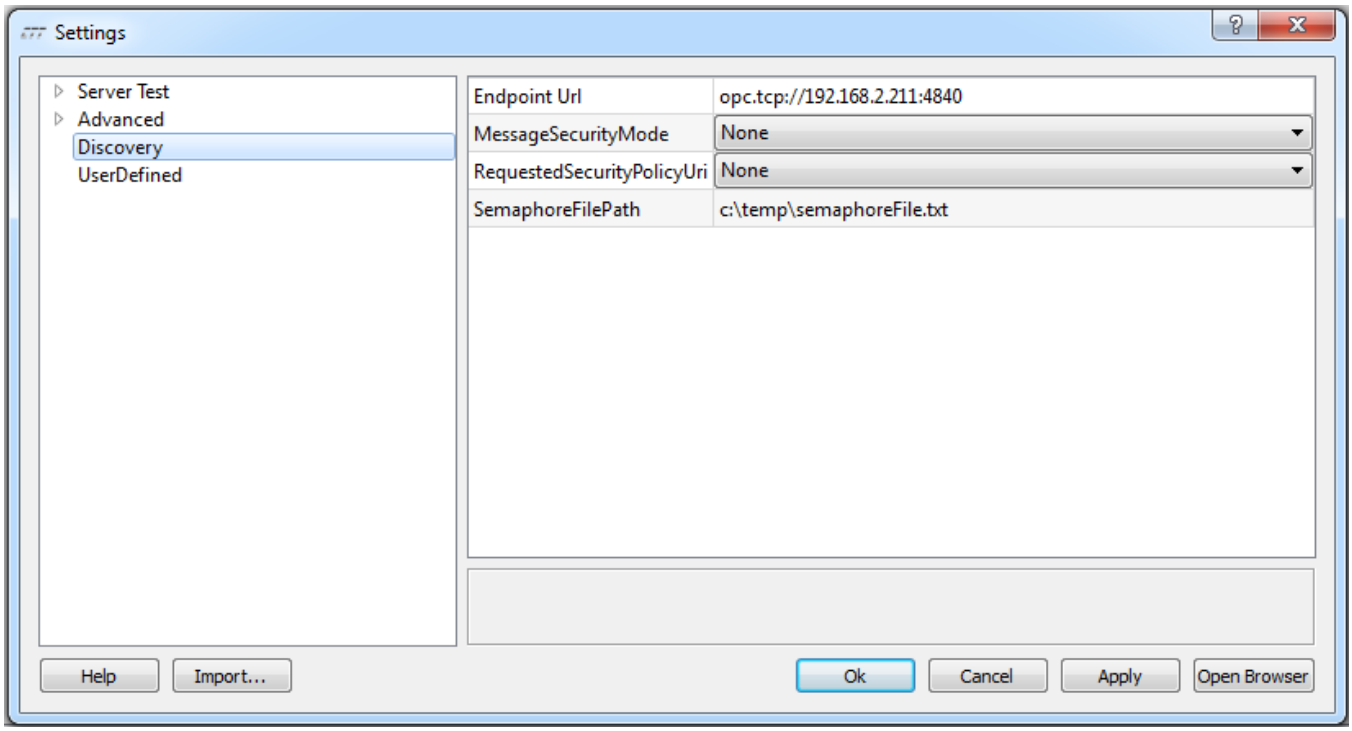
Two files are provided with the example project:

- compliance.ctt.xml
- compliance.selection.xml

In order to use these files follow the steps below:

1. Install the CTT and create a new server test project.
2. Shutdown the CTT.
3. Replace the "<project name>.ctt.xml" configuration file in your test project with the file provided and add the "<project name>.selection.xml" file to your project folder by renaming the files provided from "compliance.*" to "<project name>.*".
4. Restart the CTT and open the existing project that you had previously created.
5. Open the settings tab and change the Server URL and Endpoint URL to match your server as shown in the dialogs below.





6. Start a test run.

The CTT is very sensitive to time synchronisation so ensure the server and client systems are accurately synchronized using NTP or another mechanism.

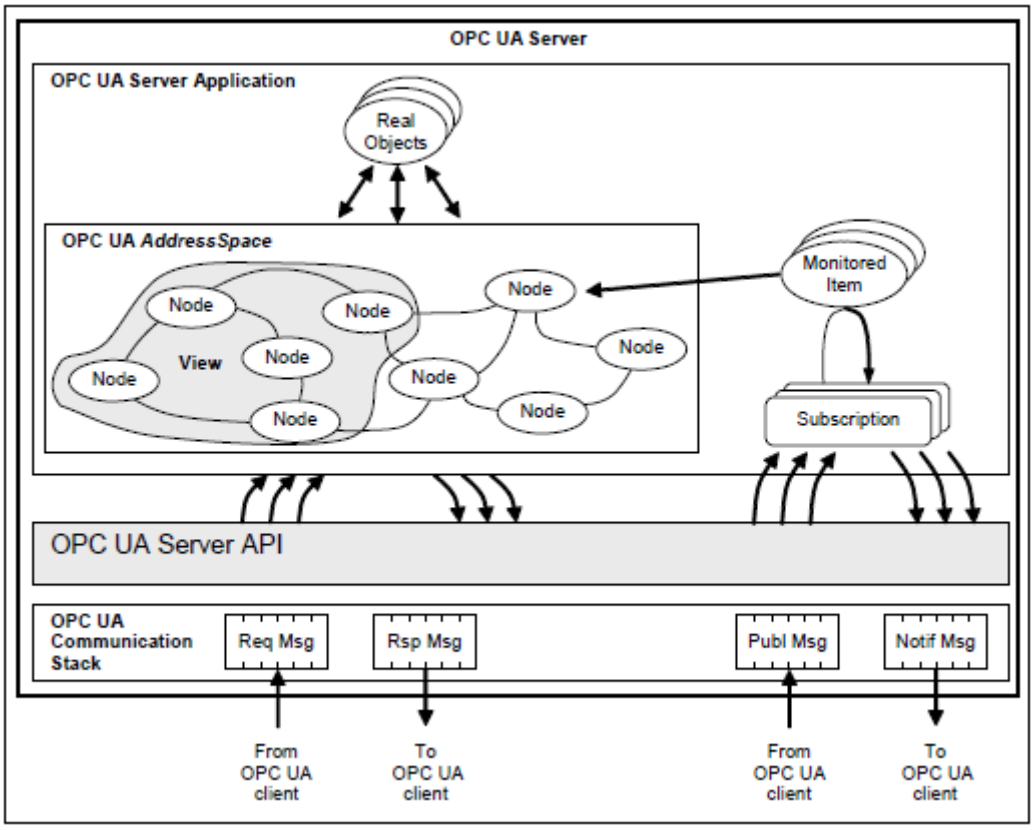
What you need to know about OPC UA

This section introduces the reader to the key OPC UA concepts required to model process data in the server. This is not meant to be an exhaustive and complete description of the standard (the standard itself is hundreds of pages long).

OPC-UA Server Concept

OPC-UA is a communications and data modelling standard for the exchange of information over TCP/IP networks. The data exchange is performed between client and server applications. Client applications are typically consumers of process data such as visualisation tools like Human Machine interfaces (HMI) and SCADA clients (Supervisory Control And Data Acquisition). 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.

The diagram below (an extract from the standard documentation) represents an OPC-UA Server and shows its key components.



Real objects are process or data variables that the server wishes to expose. These objects are mapped to *nodes* in an address space. Each node contains *attributes*. These attributes contain information pertaining to the node, the most important of which is the value attribute, representing the value of the real object.

The *AddressSpace*, or a subset of the address space called a *view* can be explored by OPC-UA clients. When a client identifies a node attribute that it wants to read or write, it can do so by

sending 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.

Address Space

The address space is the heart of the 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 that variable, etc.

Properties are standard nodes that a particular type of node references or contains. For example, an *AnalogItem* node has an *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 will be explained in this document.

Folders

Folders are a type of node that contains other nodes. When we say contains other nodes, in reality they reference other nodes but we can think of them in terms of containing other nodes like a folder contains documents.

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 particular 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 via different views with different read and write permissions to ensure that the end user is presented with data in a way that makes sense to them.

The set of nodes in different views can be overlapping, that is to say that different views can represent the same data in different ways or represent different data, or a combination of both.

Variables

Variables are the most important nodes in the address space as they are the nodes that expose your process data to the client. There are many types of variable node and some node types can

themselves contain many different types of 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. When we talk about variable types, we mean the different type 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 attribute, 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 can be of a variety of data types including strings and byte arrays.

If you want to quickly learn how to configure an OPC-UA Server with *BaseDataVariables* we would recommend that you skip ahead to the next section 3.4 and then on to our quick start guide which will show you how. If you want to first understand Data Access types in detail, please read on.

Data Access Variables

Data Access (DA) variable types are a more structured variable type with more features than *BaseDataVariables*. It is perfectly valid to use only *BaseDataVariables* to expose all of your data if that is the functionality you require. DA types are only necessary when you need the extra features the types provide.

Also, DA types require more storage in the server so you can expose less DA types than *BaseDataVariables* for a specific amount of RAM. If you are working in a highly resource constrained environment only use DA types when you need to. 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 as follows:

- *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 particular 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

[1] Server localization is the creation of a server that exposes it's human readable strings in a variety of languages depending on the requirements of a particular client. Localization is fully supported by the Device Server SDKs but is completely optional. If your customers need it you can provide it, but there is no obligation to provide it.

the client. A server is expected to silently round any value with more precision than it supports.

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 normal operation. It is intended for such use 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 as follows:
  - *namespaceUri*.
    This string specifies the organisation or standards 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 metre).
  - *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.

Sessions and Requests

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

From your perspective sessions are handled transparently by the library. All communication activity between a client and your 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, the client establishes a *subscription* with the server. 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 dequeues 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 your perspective subscriptions and monitored items are handled transparently by the library. Whenever the library needs to read a data value in order to perform sampling of a monitored item it will execute the read callback in order 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.