

**Adda (Advanced data access) -interface
documentation for OPC COM DA Kit or XML
DA Kit users.**

VERSION 0.8 DRAFT

| Customer:

Public		Registered in VTT publications register JURE	
Confidential until / permanently			
Internal use only			
Title Adda (Advanced data access) -interface documentation for OPC COM DA Kit or XML DA Kit users.			
Customer or financing body and order date/No.		Research report No.	
Project		Project No.	
Author(s) Jyrki Peltoniemi		No. of pages/appendices 24 /	
Keywords Simulation control, OPC, OPC XML DA			
Summary This document describe interface to be implemented to utilize OPC XML DA or OPC COM DA Kit. These two libraries are used to provide standardised data access interface mainly for simulators, but also other data sources like automation systems.			
Date Espoo 25 March, 2011			
N.N		Jyrki Peltoniemi Research Scientist	Checked
Distribution (customers and VTT): Siemens, Framatome, Fortum Nuclear Services, Fortum Power and Heat			
<i>The use of the name of VTT in advertising, or publication of this report in part is allowed only by written permission from VTT.</i>			

Version history

Numb	Date	Author	Comment
0.1	22.12.05	Jyrki P.	
0.2	1.2.06	Jyrki P.	
0.3	22.4.06	Jyrki.P.	
0.6	8.5.06	Jyrki P.	Item and branch properties clarification added.
0.7	29.6.09	Pasi L.	Replay and SwitchCheck added
0.8	25.3.11	Tuomas M.	Free added.

Table of contents

1	Introduction	5
2	Interface overview	6
3	Utility functions	7
3.1	Registered functions.....	7
3.1.1	addasRegLogMessage	7
3.1.2	addasRegOpenConnection	7
3.1.3	addasRegReadyToQuit.....	8
3.1.4	addasRegGetOPCProperties	8
3.1.5	addasRegGetLastError	8
3.1.6	addasRegFree.....	8
3.2	Event functions.....	9
3.2.1	adxmlsInit	9
3.2.2	adxmlsExit.....	9
4	Browsing.....	9
4.1	Basic browsing	9
4.1.1	addasRegGetBranches	9
4.1.2	addasRegGetLeaves.....	10
4.1.3	addasRegGetLeafId	10
4.1.4	addasRegGetLeafIdType	10
4.1.5	addasRegRootName (deprecated).....	11
4.1.6	addasRegIsFlat	11
4.1.7	addasRegSeparation.....	11

4.2	Item properties	11
4.2.1	addasRegQueryAvailableProperties	11
4.2.2	addasRegGetItemProperties.....	12
4.2.3	addasRegLookupItemIds	13
4.3	Branch properties	13
4.3.1	adxmlsRegQueryAvailableBranchProperties.....	13
4.3.2	adxmlsRegGetBranchProperties	14
5	Data Access	14
5.1	Registered functions.....	14
5.1.1	addasRegAddGroup.....	14
5.1.2	addasRegDelGroup.....	15
5.1.3	addasRegChangeFrequency.....	15
5.1.4	addasRegGetFrequency	15
5.1.5	addasRegAddItems	15
5.1.6	addasRegRemoveItems	16
5.1.7	addasRegCheckItemIds	17
5.1.8	addasRegWrite.....	17
5.1.9	addasRegQueryDataChanged	18
5.2	Event functions.....	18
5.2.1	addasConfigurationChanged.....	18
5.2.2	addasDataChanged	18
5.2.3	addasDataChanged2	19
6	Simulation Control and other extensions	19
6.1	Basic simulation control.....	19
6.1.1	adxmlsRegRun.....	19
6.1.2	adxmlsRegSetSpeedAndRun.....	19
6.1.3	adxmlsRegStop	20
6.1.4	adxmlsRegLoad	20
6.1.5	adxmlsRegLoadWithBasetime	20
6.1.6	adxmlsRegSave	20
6.1.7	adxmlsRegLoadState	20
6.1.8	adxmlsRegSaveState.....	21
6.2	Commands and events	21
6.2.1	adxmlsLaunchCommands	21
6.2.2	adxmlsReverseCommands	22
6.2.3	adxmlsReadCommands	22
6.2.4	adxmlsRegSubscribeEvents.....	22
6.2.5	adxmlsRegSubscribeCancel	23
6.2.6	adxmlsEventOccured	23
6.3	Simulation control events	23
6.3.1	adxmlsStateChange	23
6.3.2	adxmlsRegGetState	24

6.4 Synchronization.....	24
6.4.1 adxmlsContinue.....	24
6.5 System control.....	24
6.5.1 adxmlsStartUp.....	24
6.5.2 adxmlsShutDown	24
6.5.3 adxmlsGetSystemInformation	25
6.5.4 adxmlsGetModelInformation.....	25
6.5.5 adxmlsGetStatusInformation	25
6.5.6 adxmlsRegReplay	25
6.5.7 adxmlsSwitchCheck	26
References	26

1 Introduction

OPC XML DA (1.01) specification and OPC DA Custom Interface (2.05) specification are data access specifications that are defined in OPC Foundation (www.opcfoundation.org). Both of these specifications define how to get and set data from real time devices using client-server communication paradigm and Ethernet based communication. OPC DA specification is using Microsoft COM (Component Object Model) technology and OPC XML DA specification is using web services technology. Both of these specifications provide quite similar functionality, even if the data transfer protocols are different. OPC DA uses binary transportation protocol and XML DA uses SOAP and http to transport data from server to client. OPC DA service interface is defined using Microsoft IDL (Interface definition language) and XML DA service interface using WSDL (Web Services Description Language). This document specifies how to utilize OPCDAKit.dll and XMLDAKit.dll to get these two communication protocols implemented. The reader of this document is supposed to be familiar with either (or preferably both) OPC DA Custom interface specification or OPC XML DA interface specification.

2 Interface overview

XMLDAKit.dll and OPCKit.dll are libraries that provide implementations of OPC XML DA and OPC DA Custom interface specifications respectively. The interface that have to be implemented to be able utilize either of these components is called Adda (Advanced Data Access) interface. Underlying data source (e.g. a simulator or an automation system) can link both these DLLs to implement both specifications or just either one. The interface for these two components is essentially same. For the sake of simplicity in this document we discuss only the interface that is required to implement XML DA specification. It is quite straightforward to utilize also OPCKit after Adda-interface has been implemented for XML DA Kit or wise versa (Figure 1).

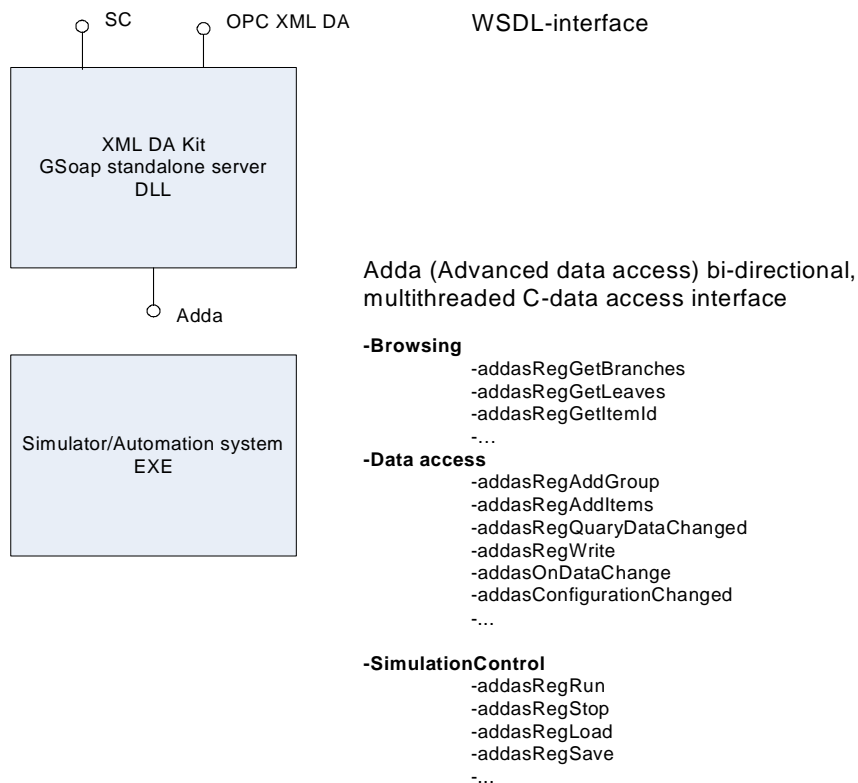


Figure 1 Overview of Design.

Functionality that is provided by the XML DA Kit is divided into two separate interfaces (WSDL s). OPC XML DA interface is specified by OPC Foundation and provides means for browsing the underlying data base of data source, reading the data from the data source and writing data to data source. SC (Simulation Control) interface is defined at VTT and specifies functions that are specific to simulator and other non-real-time systems. This interface is also defined using WSDL.

Adda is the interface that one should implement to be able to use XML DA Kit. Adda interface is here divided into three sections. Browsing is used to implement standard OPC browsing capabilities. Adda data access functions are needed to be able to implement OPC data access capabilities. Simulation control functions are needed if system is wanted to support also simulation control functionality defined in WSDL provided by VTT. These functions are documented in sections 4- 6 respectively.

Adda interface is defined using standard C. Interface is defined in three files addas.h, adopc.h and adxmls.h. The addas.h is the most important one and most functions and their documentation are found from there. The adxmls.h and adopc.h mostly define new names for the functions using pre-processor macros. Simulation control functions are also defined in adopc.h and adxmls.h. XML DA Kit supports slightly different kind of simulation control functionality than COM DA Kit. Simulation control functions that are used only by XML DA Kit are defined in adxmls.h.

The Adda-interface is a bi-directional interface. The XML DA Kit is linked from the underlying simulator. After this it is up to the simulator to register functions that are called by XML DA Kit in runtime. The most functionality is going from XML DA Kit to the simulator. There are also some event-functions that are directly called by the simulator.

Adda-interface is multithreaded library and the underlying simulator must be able to handle simultaneous calls to registered functions.

3 Utility functions

3.1 Registered functions

3.1.1 addasRegLogMessage

Kit calls to inform simulator about different events. It is up to simulator to decide what to do with these messages e.g. Apros writes these messages to log file or to console. These messages can be used later on e.g. to debugging.

Parameters:

char* aMessage: Null terminated string.

int aNro: Indicate whether the log-message describes serious error or quite normal behaviour. Serious error is 0 and normal error/message is 1. Typical serious errors are initialization errors etc.

Returns: 0.

3.1.2 addasRegOpenConnection

The first registered function to be called. This is called when some XML DA client or COM DA client first time establishes a connection. It is always the first function that is called. It must also be called if the function call before this has been addasReadyToQuit.

Parameters:

AddaModuleId aModuleId: Either COMDAKit or XMLDAKit

Returns: 0.

3.1.3 addasRegReadyToQuit

COM DA Kit calls this when last client detached from the server. XML DA is connectionless specification and due to that this function is not currently called from the XML DA Kit.

Parameters:

AddaModuleId aModuleId: Either COMDAKit or XMLDAKit.

Returns: 0.

3.1.4 addasRegGetOPCProperties

This function is only needed in COM DA Kit. Simulator that implements only XML DA Kit may return anything to all output parameters.

Parameters:

int aSynchronizationState: Dummy

int aSendAll: Dummy

float aTimeOut: Dummy

Returns: 0.

3.1.5 addasRegGetLastError

Returns the size of the last error message. If not null then apBuffer is used to return the last error message. If the buffer size isn't big enough the message is truncated. The message will always end with null (\0). This function is only used for XMLDA kit. It can be used because only one thread is allowed to execute in OPCKit simultaneously -> No changes to get old wrong error messages.

Parameters:

int aSize: Max size of buffer.

char* apBuffer: Memory that is used to get error messages (OPCKit allocates this memory).

Returns: The size of error last message.

3.1.6 addasRegFree

This function is only needed in COM DA Kit. Frees the pointer that was malloc()'ed in the simulator. (If the simulator is compiled with a different compiler than the Kit, it may be that the memory allocated in the simulator cannot be freed in the kit.) Simulator that implements only XML DA Kit may return anything.

Parameters:

void* aPointer: Pointer to the item is to be freed.

Returns: 0.

3.2 Event functions

3.2.1 adxmlsInit

First function to be called when the OPC XML DA Kit has been loaded and function-pointers have been registered. Specifies which port is used to run soap server.

Parameters:

const char* apOPCName: Name of the OPC XML DA Server. XML DA kit starts the server to localhost, no matter what. Value should be 'localhost'.

int aHttpPort: e.g. 8090 or something other. Currently there is no mechanism that could notice if current port is already in use so be careful.

AddaModuleId aModuleId: Should be always 'XMLDAKit'.

3.2.2 adxmlsExit

Not used.

4 Browsing

4.1 Basic browsing

4.1.1 addasRegGetBranches

Get branches from the browse tree position specified by the apPath. If query is done for root apPath = NULL.

Parameters:

int* apCount: Output parameter that specifies how many branches are returned.

char*** apBranches: Output parameter that contains branches (short name, not item ids)

const char* apPath: Input parameter that specifies browse position where the branches should be returned.

int aPathLen: Length of the apPath (null terminated string).

Returns: 0 if succeeded, 1 if not.

4.1.2 addasRegGetLeaves

Get leaves from the browse tree position specified by the apPath.

Parameters:

int* apCount: Output parameter that specifies how many leaves are returned.

char*** apLeaves: Output parameter that contains leaves (short name, not item ids)

const char* apPath: Input parameter that specifies where the leaves should be returned.

int aPathLen: Length of the apPath (null terminated string).

Returns: 0 if succeeded, 1 if not.

4.1.3 addasRegGetLeafId

Get leaf identifier, should identify each leaf uniquely. Leaf identifier maps to the item id. (Field of AddaDataItem structure in Data Access function. It is used to get data from the simulator. See addasAddItems function)

Parameters:

const char *apLeaf: Leaf name (Null terminated string) that Kit is interested in.

const char* apPath: The path of the leaf. Note that the path doesn't include leaf name because it is specified using apLeaf.

int aPathLen: Length of path.

char **apLeafId: Returned leaf id for this leaf. Should be unique within the server.

Returns: 0 if succeeded, 1 if not.

4.1.4 addasRegGetLeafIdType

Get leaf id types (AddaDataItem) and sizes (int) of one or more specified leaf id.

Parameters:

const int aCount: Number of leaves to which types are queried.

const char **apLeaf: Array of leaves.

const char* apPath: All leaves must be in same browse level.

int aPathLen: Length of previous.

AddaDataItem aTypes[]: Output parameter that contains types for corresponding leaves. Size of array is aCount.

int aSize[]: Output parameter that contains sizes for corresponding leaves and types.

Returns: 0 if succeeded, 1 if not.

4.1.5 addasRegRootName (deprecated)

Not in use.

4.1.6 addasRegIsFlat

Registers functions addasIsFlat. addasIsFlat should return always false. Flat namespace browsing is supported neither COM DA Kit nor XML DA Kit.

Parameters:

AddaBoolean* apFlat: output parameter that should always be false.

Returns: 0

4.1.7 addasRegSeparation

Get separation character, used between parts of a tree item name. Browse branches and leaves are separated using this character. Possible choices of separation chars are e.g. "!.:". Apros uses '!' character.

Parameters:

Returns: Separation char

4.2 Item properties

The idea behind item properties is documented in OPC DA Custom interface specification. For more detailed description see the addas.h.

4.2.1 addasRegQueryAvailableProperties

Return available property ids, descriptions and property types for given item.

See the available properties from addas.h. Simulator may implement all or some of those properties. However many of those properties are not so important and are not valid for all items. Browse will work without item properties as well.

Available properties (and corresponding descriptions and data types) are:

1 dataType [Item Canonical Data Type] [double 8byte]
(The first 4 bytes are AddaDatatype (enum) and last 4 byte size (int) (little bit ugly..)
Value field is not used when properties are returned, but the type and size fields, See the documentation of getItemProperties).

2 value	[Item Value]	[varying]
3 quality	[Item Quality]	[int, size=1]
4 timestamp	[Item Timestamp]	[int, size=1]

(In milliseconds time_t format)

5 accessRights (qualityGood = 1, qualityBad = 0)	[Item Access rights]	[int, size=1]
6 scanRate (Can be hard-coded e.g. 100ms)	[Server Scan Rate]	[float, size=1]
100 engineeringUnits	[EU units]	[string, size=1]
102 highEU	[high EU]	[double, size=1]
103 lowEU	[low EU]	[double, size=1]
104 highIR	[High Instrument Range]	[double, size=1]
105 lowIR	[Low instrument range]	[double, size=1]

Parameters:

const char *apItemId: Item id for which OPCKit wants available properties (NULL terminated string).

int *apCount: (Output parameter) specifies how many properties are available.

int **apIds: (Output parameter) Table of available item properties.

char ***apDescriptions: (Output parameter) Table of descriptions (table of NULL terminated strings) for available properties.

AddaDataType **apTypes: (Output parameter) Table of data types for available properties.

int** apSizes: (Output parameter) Table of sizes of available properties.

Returns: 0 if ok, 1 if invalid item id, 2 if some other error.

4.2.2 addasRegGetItemProperties

Gets defined item properties. Return property values for given item and success codes for every available property (true/false)

NOTE MEMORY! when AddaValue has string data or vector-valued data.

NOTE! special cases:

1. If propertyID = 1 (dataType), addasGetItemProperties fills aData[i].dataType and aData[i].size fields and doesn't use value-union.
2. If PropertyID = 2 (value), addasGetItemProperties fills aData[i].dataType and aData[i].size and aData[i].xxxxxValue fields. (Last field varies respect of the data type)

These two properties are special cases. For all other itemPropertyIDs AddaValue table contains type, size and corresponding AddaValue field subject to dataType.

Parameters:

const char *apItemId: Item id for which OPCKit wants specified properties (NULL terminated string).

const int apCount: Number of property ids in apIds table.

const int *apIds: Table of properties the OPCKit wants.

AddaValue aData[]: (Output parameter) Filled by the underlying simulator. It contains values for specified item properties. Note that some property is of type string or vector sized, then underlying system must allocated new memory for values with malloc.

int aResults[]: Result table filled by the simulator. Contains 0, if corresponding item id exists in system, 1 if not. Note that this is (for some weird reason) specified just opposite to normal behaviour!

Returns: 0 if successful, 1 if invalid item id, 2 if error was some other reason.

4.2.3 addasRegLookupItemIds

See addas.h for documentation. Not used by the XMLDAKit.

4.3 Branch properties

4.3.1 adxmlsRegQueryAvailableBranchProperties

For supporting XML DA item properties for branches. These are needed for command and command group browsing purposes. OPC XML DA specification supports properties also for branches. However OPC XML DA Kit calls these two functions only for the branches that have path that begins with string "COMMAND_GROUPS". So branch properties are only available for the Commands browsing. There are two branch properties required:

id=5000, name = "type", data type = string, size=1

id=5001, name = "category", data type = string, size = 1

Type can have two values: "Command group" or "Command". Category can have same values as AddaEventFilter. Function is only called by XMLDAKit, not by the COM DA Kit.

Parameters:

const char* apPath: Browse path of some branch.

int* apCount: Number of available branch properties.

int **apIds: Array of property ids (Currently two allowed 5000 and 5001)

char ***apDescriptions: Describes the returned branch properties.

AddaDataType **apTypes: Types for the returned properties

int** apSizes: Sizes of returned properties.

Returns: 0 if succeeded, 1 if not. ! is returned also if there is no branch properties for this branch or specified path is invalid.

4.3.2 adxmlsRegGetBranchProperties

Gets specified branch properties. Before calling this function Kit has called the function QueryAvailableBranchProperties to retrieve available properties. The GetBranchProperties function is only called by the XMLDAKit not by the COM DA Kit.

Parameters:

const char* apPath: Browse path of some branch (NULL terminated).

int aPathLen: Length of browse path.

const int apCount: Number of properties Kit wants

const int *apIds: Array of ids of branch properties

AddaValue aData[]: Array that is filled by the simulator with values of given properties. Note that for string underlying simulator allocates memory with malloc.

int aResults[]: Result codes for branch properties. 0 if failed 1 if succeeded.

Returns: 0 if succeeded, 1 if not.

5 Data Access

5.1 Registered functions

5.1.1 addasRegAddGroup

Grouping items is basically a mechanism to optimize data exchange. The group concept is similar to the OPC Group concept of the OPC DA Custom interface standard. A group has to be created before user can read or write data to the simulator.

Parameters:

AddaModuleId aModuleId: Enumeration that may have either value COMDAKit or XMLDAKit. XML DA Kit uses value XMLDAKit and OPCKit (COM) value COMDAKit. Underlying simulator uses this information when it decides whether to use either adxmls or adopcs functions to call back. (e.g. function adxmlsDataChanged vs. adopcsDataChanged).

int* aGroupId: Simulator returns unique integer handle that identifies the group. This value is later on used when group is deleted or items are added or the XMLDAKit writes one or more items in group.

Returns: Must return always 0 (succeeded).

5.1.2 addasRegDelGroup

Delete a group that has been created using addGroup function.

Parameters:

int aGroupId: Handle to group that is to be removed.

Returns: 0 if succeed, 1 if failed e.g. because group with such handle cannot be found.

5.1.3 addasRegChangeFrequency

Set or change the frequency of the group. After this, actual frequency can be queried using getFrequency function. Note that frequency is given in simulation time. This means that when simulation is running other than real time, events can be generated faster or slower than real time respectively to real time ratio of simulation. If simulation is in idle state (i.e. not running) group doesn't generate events to the XMLDAKit unless someone writes (using XMLDAKit or some other available native mechanism) to items that the XMLDAKit monitors.

Parameters:

int aGroupId: Handle to the group

double aFreq: Requested frequency of a group in seconds. May be fraction of seconds.

Returns: 0 if succeeded, 2 = Illegal frequency. 3 = Unsupported frequency

5.1.4 addasRegGetFrequency

Get the frequency of the group.

Parameters:

int aGroupId: Handle to the group

double* aFreq: (Revised) frequency of a group in seconds. Can be fraction of seconds.

Returns: 0 if succeeded, 2 = Illegal frequency. 3 = Unsupported frequency

5.1.5 addasRegAddItems

Add items to the data communication. addasAddItems function fills fields data, size, frontItem and quality in aItem structure table.

Note that aItemIds table CAN contain ids of items which are already added to the communication. Simulator may or may not create duplicate items in group. However, simulator removes all possible duplicate items with single addasRemoveItems function-call (from the specified group).

Simulator doesn't have to indicate in any way whether there already was similar item in communication in specified group.

AddasAddItems is group specific operation. Because of historical reasons, group is specified in AddaDataItems table for each item. Simulator may take the group from the first AddaDataItem and trust that all the rest items belong to the same group. XML DA Kit and COM DA Kit both use addasAddItems in this way. It is NOT allowed to call addasAddItems in a manner that a single list contains items that should be inserted to different groups. The same rule is applied to functions: addasRegAddItems, addasRegRemoveItems, addasRegWrite and addasConfigurationChanged.

AddasAddItems should call (synchronously) adxmlsConfigurationChanged(aSize, itemPtrTable) function to notify Kit that new items have been added. See addasConfigurationChanged documentation.

Parameters:

int aSize: Size of item array.

AddaDataItem[] aItemIds: Array of item AddaDataItems. In-out parameter; Kit fills the item id and the simulator fills associated type, size, frontItem and quality fields.

Returns: 0 if succeeded, 1 if not.

5.1.6 addasRegRemoveItems

Remove items from the data communication. Items must be on the given group and group is specified in a first item in AddaDataItem table (historical reasons, see the documentation of addasAddItems).

Note that aItemIds table CAN contain ids of items which are already added to the communication. Simulator may or may not create duplicate items in group. However, simulator removes all possible duplicate items with single addasRemoveItems function-call (from the specified group).

Simulator doesn't have to indicate in any way whether there already was similar item in communication in specified group.

AddasAddItems is group specific operation. Because of historical reasons, group is specified in AddaDataItems table for each item. Simulator may take the group from the first AddaDataItem and trust that all the rest items belong to the same group. XML DA Kit and COM DA Kit both use addasAddItems in this way. It is NOT allowed to call addasAddItems in a manner that a single list contains items that should be inserted to different groups. The same rule is applied to functions: addasRegAddItems, addasRegRemoveItems, addasRegWrite and addasConfigurationChanged.

AddasAddItems should call (synchronously) adxmlsConfigurationChanged(aSize, itemPtrTable) function to notify Kit that new items have been added. See addasConfigurationChanged documentation.

Parameters:

int aSize: Size of item array.

AddaDataItem[] aItemIds: Array of item AddaDataItems. Only Item id and group id fields are used.

Returns: 0 if succeeded, 1 if not.

5.1.7 addasRegCheckItemIds

Validate the existence of items. Kit fills the item id field of AddaDataItems. Front fills the AddaDataItems with right data type and size. On return the apExist table must contain 0 if the item id does not exist and 1 if it exists. This function is used e.g. in OPC XML DA Read, Write and CreateSubscription. It is used solely for item validation purposes and addasAddItems function should be used only for those items that are validated with this function. addasAddItems cannot be used for validation.

Parameters:

int aSize: Size of item array.

AddaDataItem[] apItemIds: Array of item AddaDataItems. Only Item id and group id fields are used.

int[] apExist: Result table filled by the simulator. Contains 1, if corresponding item id exists in system, 0 if not.

Returns: 0 if succeeded, 1 if not.

5.1.8 addasRegWrite

Write to one or more values in a simulator. Simulator can return immediately after it has copied the values into its own variables. apAddaItems gives the same information which is given in addasConfigurationChanged. it contains same pointers to Front data. Simulator used these pointers to identify items that Kit tries to write. apData array contains pointers to the data in OPCKIT/Adda dll. They are in same order as AddaDataItems.

Before using addasWrite XML DA Kit has to create group and add items into it. Group id is specified in the first AddaDataItem and must be same for all items in apAddaItems array (see also the documentation of addasAddItems)

If item does not anymore exists or it's attributes (iDim, charLength) has changed then its quality is AddaInvalid and this information is filled to the apAddaItems quality field by addasWrite function. For this item no write operation is done and addasWrite returns 1. If all writes succeeded then function returns 0.

Parameters:

int aSize: Size of item array.

AddaDataItem[] apAddaItems: Array of item AddaDataItems. FrontItem field is used to identify items.

void** apData: Data that is written to simulator. Data types and sizes have to match with corresponding item.

Returns: 0 if all writes succeeded, 1 if not.

5.1.9 addasRegQueryDataChanged

Function is used to force dataChanged function call as soon as possible. Used e.g. in COM DA Kit functions Device Read and Device Refresh and in XML DA functions Read and Write. Returns 0 when (adopcs) (adxmls)DataChanged has been called and returned for this group, i.e. new values are copied to the Kit (if they are changed). Simulator may call adxmlsDataChanged(adopcsDataChanged synchronously from the same thread that called addasQueryDataChanged function.

Parameters:

int aGroupId: ad(opcs/xmls)DataChanged function is called for this group.

Returns: 0 if succeeded after dataChanged has been called. 1 if fuction failed, e.g group was not found.

5.2 Event functions

5.2.1 addasConfigurationChanged

Notify that the simulator has changed the set of data items in transfer or that their data pointers, pointers to the FrontItems or qualities have changed. It is called synchronously in from addasAddItems but simulator may generate these also by itself. Typical situation when configuration may change is when new model is loaded and there exists items in data communication. In new model all items ids does not necessarily exist.

If addasConfigurationChanged call is generated because items have been removed from the simulator, AddaDataItem field frontItem is set to NULL and AddaQuality to AddaInvalid. In this case simulator should not remove Items from the communication before it has called this function and it has been returned.

Parameters:

int aNumberOfItems: Number of items in array.

AddaDataItem*[] apItems: Array that contains all those items that has changed in specific group. Function should be called in such a way that all items in array are members of same group (historical reasons see addasAddItems for more details). All group ids are filled and should be same.

Returns: 0 if succeeded after dataChanged has been called. 1 if fuction failed, e.g group was not found.

5.2.2 addasDataChanged

Notify that data has changed in simulator. Function is group-specific like addasConfigurationChanged. Typical implementation of this function is to loop all items in the group and copy the values from simulator to Kit's cache if the value has changed respect of the previous call. Function uses data pointers that simulator has given to kit using addasConfigurationChanged function.

During fuction execution Kit accesses the database of simulator, however it is not a good idea to hold any kind of resourses reserved when addasDataChanged function is called. This may result a deadlock. E.g. it is typical situation that addasWrite has to be able executed while addasDataChanged is on.

It is up to Kit to determine whether the values are changed or not. Simulator just notifies that this group has reached the frequency of group and new data is available.

Parameters:

double aCLTime: Simulation time in seconds. Used in time stamps in Kit. Kit forms the timestamps of items by adding simulation time to server start time.

int aGroupId: Group id for the group. This implies that simulator must call DataChanged function to all groups the Kit has added to communication.

Returns: 0 if succeeded after dataChanged has been called. 1 if fuction failed, e.g group was not found.

5.2.3 addasDataChanged2

Function is implemented only in COM DA Kit. It is solely for optimization purposes for situations where simulator knows exactly which items have changed.

6 Simulation Control and other extensions

6.1 Basic simulation control

6.1.1 adxmlsRegRun

Not currently used in XML DA Kit. In XML DA Kit the function SetSpeedAndRun is used instead of this.

Parameters:

double aTime: Simulation time in seconds that specifies how long the simulation should be in simulation time scale.

Returns: 0 if succeeded, 1 if fuction failed.

6.1.2 adxmlsRegSetSpeedAndRun

Used only in XML DA Kit. Starts simulation run with specified simulation speed. Simulation speed is given using real time ratio.

Parameters:

double aSpeed: Speed that simulation should be run. After simulation has started it is run as long as it is stopped e.g. using function stop. -1 is set to speed if existing speed should be used.

Returns: 0 if succeeded, 1 if failed.

6.1.3 adxmlsRegStop

Stop the simulation. If simulation is already in stopped state nothing happens.

Parameters:

Returns: 0 if succeeded, 1 if failed.

6.1.4 adxmlsRegLoad

Load a new model.

Parameters:

char* aModel: Name of the model to load. Semantics of aModel is not specified. It may be e.g. file name of snap file from where a new state is loaded.

Returns: 0 if succeeded, 1 if failed. Failure may be e.g. due to the invalid model name or some application specific loading error.

6.1.5 adxmlsRegLoadWithBasetime

Load a new model and give a base time. Note that e.g. Apros doesn't use base time, but only simulation time when it sends data change events. XML DA Kit adds this base time to simulation time when it forms the timestamps. The current version of OPCKit also loses milliseconds from given base time.

Parameters:

char* aModel: Name of the model to save. Semantics of aModel is not specified. It may be e.g. file name of snap file where the model is to be saved.

AddaTimestamp aBasetime: Time structure defined in adxmls.h

Returns: 0 if succeeded, 1 if failed.

6.1.6 adxmlsRegSave

Save the whole model.

Parameters:

char* aModel: Name of the model to save. Semantics of aModel is not specified. It may be e.g. file name of snap file where the model is to be saved.

Returns: 0 if succeeded, 1 if failed.

6.1.7 adxmlsRegLoadState

Load the state.

Parameters:

char* aState: Name of the state to load. Semantics of aState is not specified. It may be e.g. file name of snap file from where a state is to be loaded.

Returns: 0 if succeeded, 1 if failed.

6.1.8 adxmlsRegSaveState

Save the state model. The state of the model does not necessarily contain all static information in model (contrary to adxmlSave). This may lead to significantly faster execution than achieved using save.

Parameters:

char* aState: Name of the state to save. Semantics of aState is not specified. It may be e.g. file name of snap file from where a state is to be saved.

Returns: 0 if succeeded, 1 if failed.

6.2 Commands and events

6.2.1 adxmlsLaunchCommands

Commands, that may or may not cause events, are executed using Launch command. Launching is done based on command id. Currently there is no way to browse all commands available in system using this interface. Browsing is done using the normal OPC browse interface.

Each command may or may not have command properties (AddaCommandProperty). Each command property has name and value. Name of the command property should be unique within command. Kit may give one or more command properties when it launches the command. It doesn't have to specify all of them. The semantics are the command properties that are not given, used using old (or default) values, is not specified here. Command property values can be currently any scalar AddaDataTypes (AddaShort, AddaInt, AddaFloat, AddaDouble, AddaChar, AddaString).

Parameters:

int aNumberOfCommands: Number of elements in aCommands array.

AddaCommand aCommands[]: Command array to be launched. AddaCommand structure is declared in adxmls.h. The state of command is not meaningful in launch function. In launch operation, the state of the command is always set to true.

int aErrors[]: For each command there is corresponding result code. Result code is 1 if reversing command succeeded 0 if not. If e.g. some command parameters were invalid or their data types don't match, those command parameters may silently be discarded if command could be reversed regardless of that. So success (indicated by 1) may be partial. It is up to underlying system to decide when reversing operation failed.

Returns: 0 if succeeded, 1 if failed.

6.2.2 adxmlsReverseCommands

Commands are executed launched using adxmlsLaunchCommand. adxmlsReverseCommands is used to set command trigger value to false. One or more commands may be specified. If command to be reversed is not found or cannot be executed corresponding error is set.

Parameters:

int aNumberOfCommands: Number of elements in aCommands array.

AddaCommand aCommands[]: Command array to be reversed. AddaCommand structure is declared in adxmls.h . The state of command is not meaningful in reverse function. In reverse operation, the state of the command is always set to false.

int aErrors[]: For each command there is corresponding result code. Result code is 1 if reversing command succeeded 0 if not. If e.g. some command parameters were invalid or their data types don't match, those command parameters may silently be discarded if command could be reversed regardless of that. So success (indicated by 1) may be partial. It is up to underlying system to decide when reversing operation failed.

Returns: 0 if succeeded, 1 if failed.

6.2.3 adxmlsReadCommands

Read command is used to retrieve command by command name.

If command has command properties, these are also returned. Using the returned command user of Kit can launch or reverse it.

Parameters:

int aNumberOfCommands: Number of elements in aCommandIds array

const char** aCommandIds: Command Id array that uniquely specifies command that is to be read.

AddaCommand* apCommands[]: Output parameter. Simulator fills the commands. If command was not found or error occurred, NULL is filled and corresponding aErrors array element is set to 0. Commands include also command properties if such are available. AddaCommand and AddaCommandProperty structures are declared in adxmls.h.

int aErrors[]: For each command there is corresponding result code. Result code is 1 if reading command succeeded, 0 if not. Failure may be e.g. due to the invalid or unknown command id.

Returns: 0 if succeeded, 1 if failed.

6.2.4 adxmlsRegSubscribeEvents

Events can be subscribed using adxmlsSubscribeEvents. List of event filters are used to specify what events the Kit is interested in.

Parameters:

int aSubscriptionHandle: Sever specifies unique handle that is used when events are sent or event subscription is cancelled.

double aFreq: 0 if observation of events should be as frequent as possible. Otherwise it is time in seconds (simulation time).

int aNumberOfEventFilters: Number of elements in aEventFilters array.

AddaEventFilter aEventFilters[]: Event filters that are used to define that only these kind of items are generate events to Kit.

Returns: 0 if succeeded, 1 if failed.

6.2.5 adxmlsRegSubscribeCancel

Removes the specified subscription

Parameters:

int aSubscriptionHandle: Handle of subscription to be removed.

Returns: 0 if succeeded, 1 if failed.

6.2.6 adxmlsEventOccured

Notify that one or more events have occurred. It is implementation detail whether the observation really is event based or is it implemented using looping.

Parameters:

int aSubscriptionHandle: Handle of subscription.

int aNumberOfEventsChanged: Number of events in aChangedEvents array.

AddaEvent** aChangedStates: Array of event that has occurred.

6.3 Simulation control events

Simulation control events are events that communicates the state of the underlying simulator. Possible states are: AdopcsStopped, AdopcsRunning, AdopcsStoppedRecording, AdopcsRunningRecording, AdopcsStoppedPlaying, AdopcsRunningPlaying, AdopcsBusy and AdopcsUnknown. These are defined in structure AdopcsSimulationState in adopcs.h.

6.3.1 adxmlsStateChange

Notify the Kit that simulation or recording state has changed. Time is a current simulation time from the beginning of the simulation in seconds.

Parameters:

double aSimulationTime: Simulation time in seconds.

AdopcsSimulationState aSimulationState: New state

Returns: 0.

6.3.2 adxmlsRegGetState

Get the current state.

Parameters:

double* aTime: Output parameter. Current simulation time.

AdopcsSimulationState* aSimulationState: Output parameter. Current state.

6.4 Synchronization

6.4.1 adxmlsContinue

Notify the simulator that it can continue execution. Note! Not used in Apros.

Parameters:

Returns: 0 if succeeded (should always), 1 if failed.

6.5 System control

6.5.1 adxmlsStartUp

Start up the specified components. No Apros implementation. Function is for distributed components only. I.e. if behind this service interface exists multiple software components that are controlled via this interface.

Parameters:

const char* aComponentHandle: Unique handle (NULL terminated string) that identifies the component that Kit wants to start.

int aInitialize: If initialise is true, the component is reinitialised instead of restarting.

Returns: 0 if succeeded, 1 if failed. Failure may be due to invalid component handle or some other internal reason.

6.5.2 adxmlsShutDown

Shut down the specified component.

Parameters:

const char* aComponentHandle: Unique handle (NULL terminated string) that identifies the component that Kit wants to shut down.

Returns: 0 if succeeded (should always), 1 if failed.

6.5.3 adxmlsGetSystemInformation

Get the system information.

Parameters:

int* numberOfSystemInformations: Output parameter. Number of system information structures in result table.

AddaVersionInformation** aSystemInformation: Output parameter. Array of version information structures. Each version information structure may contain one or more sub version information structures and result may hence be hierarchical.

Returns: 0 if succeeded. 1 if failed.

6.5.4 adxmlsGetModelInformation

Get the model information.

Parameters:

int* numberOfVersionInformations: number of returned model infos.

AddaVersionInformation** aVersionInformation: One or more model infos. Each model info structure may contain zero or more child model infos.

Returns: 0 if succeeded. 1 if failed.

6.5.5 adxmlsGetStatusInformation

Return components under this interface.

Parameters:

int* aNumberOfComponents: Output parameter that specifies how many components is “under” this interface.

AddaComponent** aComponents: Output parameter. Array of components available behind of this interface. Each component may have one or more sub components. ie. the component structure may be hierarchical.

Returns: 0 if succeeded (should always), 1 if failed.

6.5.6 adxmlsRegReplay

Sets the replay mode of the component to aMode. Mode is given as string.
Typically value of the mode is agreed between server and client implementations.

Parameters:

char* aMode: Replay mode. This is agreed between server and client. In some applications “Station” is used to implicate that simulation control client takes care of the replay actions that this is solely information that we’re in replay mode.

Returns: 0 if succeeded (should always), 1 if failed.

6.5.7 adxmlsSwitchCheck

Request switch check to be done. This function is useful only when server supports hardware switches. Idea is that this function returns report that tells differences between switch positions and simulation model.

Parameters:

char** aDescription: Output parameter that summarizes the results.

int* lineCount: Putput parameter that tells how many lines are in aLine attribute.

char*** aLine: Output parameter, that contains a line corresponding every contradiction between the state of the hardware and software. It is good idea to generate lines so that it contains 1st the name of the contradictory device, then value in hardware and value in software separated by comma.

Returns: 0 if succeeded (should always), 1 if failed.

References

1. OPC XML Data Access specification.
2. OPC DA 2.05 Custom interface specification.